Combinatorial Problem Solving Using Randomized Dynamic Tunneling on A Production System*

Yasusi Kanada
Tsukuba Research Center, Real World Computing Partnership
Takezono 1-6-1, Tsukuba, Ibaraki 305, Japan
E-mail: *kanada@trc.rwcp.or.jp*, WWW: *http://www.rwcp.or.jp/people/yk/*

Abstract

Levy and Montalvo, Yao, and Shima individually proposed tunneling algorithms. The tunneling algorithms employ analogy to tunnel effect in physics, and are used to optimize continuous systems. The present paper proposes a method of solving combinatorial problems using a type of randomized dynamic tunneling technique. This method is based on a computational model called CCM. CCM* is an extended version of the Chemical Casting Model (CCM). CCM was proposed by the author toward developing a method of solving open and incompletely-specified problems that may change while being solved, using self-organizing computation.*

The 0–1 integer programming problem is solved using CCM with a very simple rule and an evaluation function. CCM* allows us to escape from local maxima by composing the rule dynamically and randomly. This cannot be done by using the original production rule as is. Our experiments show that approximate solutions can be found more rapidly by CCM* than by using a branch-and-bound method in the case of 0–1 integer programming.*

1. Introduction

Most combinatorial optimization problems (COPs) in Operations Research and constraint satisfaction problems (CSPs) in Artificial Intelligence are classified as NP-hard or NP-complete problems. Thus, these difficult problems are believed to be unable to find their exact solution in polynomial time. However, these problems are usually static. Constraints and evaluation functions, or objective functions, included in these problems are not varied while solving them. Real world problems are not necessarily expressed by such static constraints and evaluation functions. New information may be found while solving problems, and preexisting information may be dynamically changed by environmental change. Although such real-world problems have similarity to control problems in the respect that information varies as time goes on, the above problems are much more complex because there are both quantitative changes and qualitative changes.

Kanada [Kan 92, Kan 94] proposed a computational model called CCM, which aims to solve such problems by using self-organizing or emergent computation [For 91]. Complete information for solving the problems cannot be grasped beforehand in such situations. However, conventional combinatorial problem solving methods, including branch-and-bound methods, simulated annealing, evolutionary computation, and so on, are usually based on complete information. These methods require an objective function, which is defined using global information, to be specified completely. Thus, these methods are weak when the environment changes. CCM was developed for computation based on local and partial or incomplete information, and is based on a production system [For 81]. Production systems are often used for developing expert systems or modeling human cognitions, but CCM differs from conventional production systems in two respects. Firstly, evaluation functions, called *local order degrees*, are computed using only local information. Secondly, stochastic control or randomized ordering of rule applications is applied. Production rules and local order degrees are computed using only local information.

CCM has been applied to classical CSPs and COPs, which are closed and fully-informed. CCM has also been applied to the *N*-queens problem [Kan 94], coloring and fuzzy coloring of graph vertices or maps [Kan 95], and the traveling salesperson problem.

The present paper explains a method of solving combinatorial problems using CCM*, an extended version of CCM, in which a mechanism of tunneling [Lev 85, Yao 89, Shi 93] is added. This method makes it possible to escape from local maxima, which cannot be escaped from by applying the original production rule, by composing a new rule from the given rules dynamically and randomly. CCM* is explained in Section 2. A method of solving COPs is briefly explained in Section 3. A

* A previous *Japanese* version of this paper was presented at the Technical Group on Softeare Engineering in SICE (the Society of Instrument and Control Engineers) in 1994.

detailed method and the results of its application to 0–1 integer programming problems are described in Section 4. Section 5 gives the conclusion and mentions future work, including extensions to dynamic combinatorial problems.

2. Computational Model CCM*

This section explains computational model CCM and its extended version, CCM*.

2.1 Chemical Casting Model

The system components of CCM are shown in **Figure 1**. CCM is a computational model based on a production system, and is often used for developing expert systems. The set of data to which the rules apply is called the *working memory*. The collection of rules and functions that determine the behavior of the system, which is usually called a program, is called a *caster* in CCM.[1]

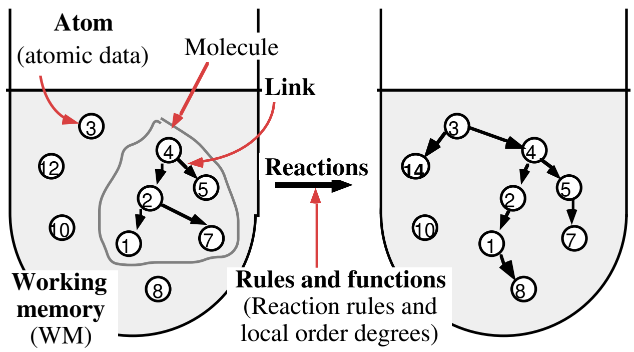

Figure 1. The elements of CCM

A unit of data in the working memory is called an *atom*. An atom has a type and an internal state, and may be connected to other atoms by *links*. Links are analogous to chemical bonds, with the difference that chemical bonds have no direction but links may have directions. Any discrete data structures such as lists, trees, graphs or networks can be represented using atoms and links.

A caster consists of reaction rules and local order degrees. *Reaction rules* change the state of the working memory locally. "Locally" means that the number of atoms referred by a reaction rule is small.[2] The reaction rules are written as so-called forward-chaining production rules [For 81], such as chemical reaction formulae or as rules used in expert systems. The syntax of reaction rules is as follows:

$$\text{LHS} \rightarrow \text{RHS}.$$

The left-hand side (LHS) or "reactants," and the right-hand side (RHS) or "products" are sequences of patterns.

For example, the following reaction rule, which is a rough sketch, simulates generation of water from oxygen and hydrogen:

$$\text{H-H, O-?} \rightarrow \text{H-O-H, ?}$$

(This approximately means $H_2 + \frac{1}{2} O_2 \rightarrow H_2O$).

There are four patterns on each side of the rule: two H's, an O, and a question mark, which is a variable that means an unknown atom. An H matches an atom of oxygen type, an O matches an atom of hydrogen type, and a variable matches an atom of any type in the working memory.

The reaction rule can be activated when there is a set of atoms that matches the LHS patterns. If the reaction rule is activated, the matched atoms vanish and new atoms that match the RHS patterns appear.[3] Just one reaction rule is enough for solving a simpler problem such as the graph vertex coloring problem or the 0–1 integer programming problem, which are described later. There will be two or more reaction rules in more complex systems, in which there are two or more ways of changing atoms.

Local order degrees (LODs) are a type of evaluation functions (objective functions). LODs express the degree of local "organization" or "order." They are defined using only local information, and the user defines that they have a larger value when the local state of the working memory is better. An LOD may be regarded as a negated *energy*. For example, it is analogous to bonding energy in chemical reaction systems.

A reaction takes place when the following two conditions are satisfied. Firstly, there is an atom that matches each pattern in the LHS. Secondly, the sum of the LODs of all the atoms that concern the reaction, i.e., that appear on either side of the reaction rule, is not decreased by the reaction. Reactions repeatedly and stochastically (or randomly) occur while the above two conditions are satisfied by any combination of a rule and atoms. The system stops when such a combination is exhausted. However, reactions may occur again if the working memory is modified because of changes of the problem or the environment. Thus, the system using CCM can solve the dynamical problems mentioned in the introduction.

In general, there may be two or more combinations that satisfy both conditions at once. There are two possible causes that generate multiple combinations. One cause is that there are two or more collections of atoms that satisfy the LHS of a reaction rule. The other cause

[1] The word "program" is not used in CCM, because "program" means a whole and complete plan. We use "caster" instead.

[2] Because (physical) distance has not yet been introduced in CCM, as it has in systems such as a chemical reaction system, "locally" does not mean the distance is small.

[3] So an atom may be modified, bonded to another atom, removed, or created by a reaction rule.

is that there are two or more reaction rules, under which there are atoms that match the patterns in the LHS. In each case, the order of the reactions, or the selection order of such combinations, and whether they occur in parallel or sequentially are determined stochastically.

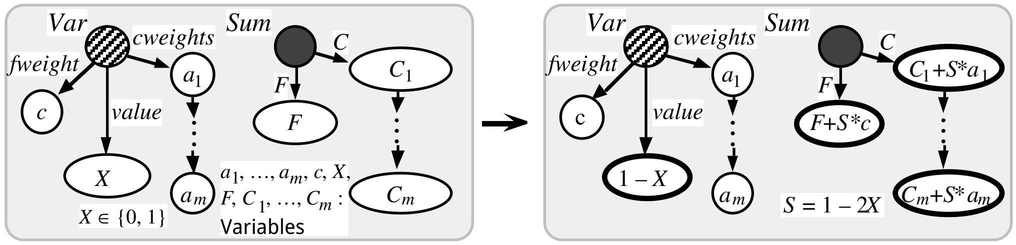

Figure 2. The reaction rule for 0–1 IPPs

Thus, the behavior of the system is determined by *coevolution*, or emergent and collective behavior, of dynamically constructed parts: the units of reactions or sets of atoms and a rule.

2.2 CCM* — an extended version of CCM

The above mechanism has the following drawbacks.

(1) A CCM-based system may easily be trapped by a local maximum. For example, the above water-generating rule might not be activated because it generates an unbonded oxygen atom, whose energy must be very high (or whose LOD must be very low).

(2) Computation in a CCM-based system may sometimes be very inefficient, because it refers only local information, i.e., only a few objects, and so it may cause reactions that do not help solvethe problem and may waste CPU time.

These drawbacks can be remedied by introducing a type of tunneling method. If reaction rules are applied two or more times without calculating the sum of LODs in between, both local maximum and inefficiency can be avoided. This process is nearly equivalent to a reaction caused by a reaction rule that is a composition of the original rules twice or more. The composition takes place dynamically and randomly. This extended version of CCM is called CCM*.

As an example of (1), if the water-generating rule is applied twice, two water molecules are generated and no single oxygen atom is generated:

H-H, O-O, H-H → H-O-H, H-O-H.

Thus the high energy state can be avoided. A detailed version of CCM* for a COP is described in the next section. The reason that this method can be regarded as a tunneling method is also mentioned in the next section.

3. Solving Combinatorial Problems Using CCM*

The present section describes a system that solves an optimization problem. This system solves the 0–1 integer programming problem, which is an example of a COP.

Integer programming problems (IPPs) are constrained linear programming problems in which values of variables are limited to integers. Real-valued linear programming problems can be solved by efficient algorithms such as the Simplex method, which is widely used, or Karmarker method, which is usually slower than the Simplex method if the problem is not very large, but which solves the problems in polynomial order time. However, IPPs are NP-hard. Thus, the scale of solvable problems is limited, although better heuristics for branch-and-bound methods are known.

The following type of IPPs, in which the variable values are restricted to 0 or 1, are called 0–1 integer programming problems (0–1 IPPs):

Maximize objective function $F = \sum_{j=1}^{n} c_j x_j$, subject to

$\sum_{j=1}^{n} a_{ij} x_j \le b_i$ ($i = 1, 2, \ldots, m$), where x_j ($j = 1, 2, \ldots, n$) are variables, $x_j \in \{0, 1\}$, and a_{ij} , b_i , and c_j are constants.

This problem is solved using CCM as follows. One of the n variables is selected randomly, and its value is switched to the other value if and only if this switching makes the objective function value (F) better. This process is repeated until no more switching will occur.

The content of the working memory, the reaction rule, and the LOD are explained below. The working memory contains an object, *Sum* (of type sum), which contains the objective function value: F, a list of the values of the LHSs of the constraints: C, and a list of the values of the RHS of them: B. The ith element of C (or B) is written as C_i (or B_i). A very simple reaction rule shown in **Figure 2** is given. This rule selects a variable *Var* of type var, and changes its value X to $1 - X$. This means that if the value is 0 it is changed to 1, and if it is 1 then it is changed to 0. Then the rule updates the values of the objective function and the LHSs of conditions in *Sum*.[1]

An LOD is defined only for *Sum*, or for objects of type sum.

$$Os(Sum) = \begin{cases} Sum.F & \textbf{if } Sum.C_i \le Sum.B_i \textbf{ for all } i \in \{1, 2, \ldots, m\} \\ -\infty & \textbf{otherwise} \end{cases}$$

An LOD is not defined for variables: the LOD for objects of type var is always 0.

When the above system is activated, a reaction occurs only when the objective function value, which is equal to the total of the LODs, increases by the reaction. Thus, the above method realizes a simple hill-climbing method, although it does not necessarily move toward the steepest slope. That means that, although the above system works on local information, or refers only a few objects in each reaction, the global computation process becomes hill-climbing. This is quite different from systems for solving CSPs, such as graph vertex coloring. Kanada [Kan 94] named such hill-climbing systems *cooperative systems*, and named other types of systems *conflicting systems*.

Because the above system is cooperative, it is easily captured by a local maxima if the original CCM is used. It is impossible to find an optimum solution, or even to find a nearly optimum one. This problem may be solved by supplying a more elaborate rule. However, this reduces the advantage of CCM, that a combinatorial problem can be solved by a combination of a simple rule and an LOD.

A better solution is to use CCM*. If the reaction rule is applied twice or more simultaneously, the system can skip over a valley of the mean order degree (MOD), as illustrated in **Figure 3**. Here, MOD means the mean value of LODs.[2] That is why this method is called a tunneling method. Tunneling methods (algorithms) were proposed by Levy and Montalvo [Lev 85], Yao [Yao 89] and Shima [Shi 93]. The tunneling methods are motivated by the concept of a tunnel effect, and are used for optimizing continuous systems. The direction of search is determined dynamically and randomly. This method can be regarded as a symbolic version of a randomized tunneling method [Shi 93].[3]

[1] The value B, a component of *Sum*, is ommitted from Figure 5, because it is not referred explicitly in this rule.

[2] Although CCM-based systems do not compute MODs, they operate so as to increase MODs or sums of LODs stochastically. MODs can be defined in small, medium, large or any other scale.

[3] Thus, the relation between the randomized tunneling method and CCM* are similar to the relation between EP (evolutionary programming) and GA (genetic algorithms), because EP is numerical and GA is mostly symbolic.

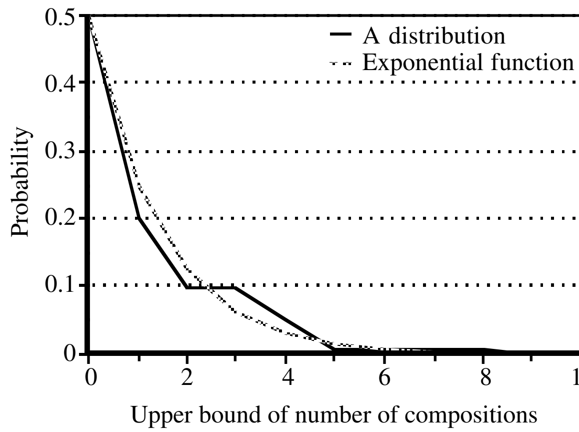

Figure 4. An example of upper bound distribution of the number of reaction

4. Application to 0–1 Integer Programming

The method of solving 0–1 IPPs shown in the previous section is described in detail, and results of experiments are shown in the present section.

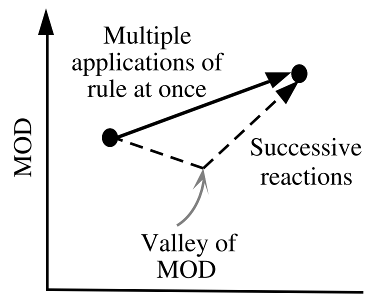

Figure 3. Skipping over a valley of MOD by symbolic tunneling

4.1 Detailed method

In an execution of CCM*, a reaction rule is interpreted in the following method:

(1) Initialization: Define the upper bound of the number of reactions, M, using a random number. The distribution of the random number is mentioned later. Define the current state of the working memory as S_0, and set i to 1.

(2) Test of reactivity: Define the state after applying the reaction rule (randomly) to the working memory in state S_{i-1} as S_i. Assert that the sequence of reactions, $a_1, a_2, \ldots, a_i$, changes state S_0 to S_i. Define the sum of the LODs of objects in state S_0, which apply to one of the reactions in the sequence, as O_0, and define that in state S_i as O_i.

(3) Reaction: If $O_0 < O_i$ (or $O_0 \le O_i$) holds, cause the sequence of reactions, $a_1, a_2, \ldots, a_i$, and return from this procedure.

(4) Preparation for next reaction: Increment i by 1. If $i \leq M$ then go to step (2) and find the next state. If $i > M$ then go to step (1) (restart this procedure from S_0).

This procedure is simplified. More exactly, the state of the working memory is updated to S_i in step (2), and it is (locally) backtracked to S_0 when going to step (1) from step (4). In addition, this procedure must be modified if there are two or more reaction rules.

The distribution of the random numbers used in Step (1) can be selected from a variety of distributions. The distribution shown in **Figure 4** is used in the experiment shown in the next subsection. This distribution is fairly close to an exponential function. However, it is not yet known whether an exponential function is really better, and why it is better if so. Anyway, the performance is not very sensitive to the distribution function in our experiments.

Known facts on the random numbers to define the upper bound of the number of reaction (the upper bound of the distribution) are as follows.

- **Search efficiency**: It is possible not to use random numbers but to use a fixed value for the upper bounds or to use an unbound number of reactions. However, experiments have shown that these methods to be inefficient at least in 0–1 IP problems. The reason is probably that to search neighbor state is probabilistically more efficient than to search distant states when the current value of the objective function is closer to its optimum value.
- **Likelyhood of getting an optimal solution**: There are cases in which the system can reach the optimum solution by a reaction rule composed μ times, but it cannot reach the optimum solution by a rule composed less than μ times. In such a case, it cannot reach the optimum solution if the number of reactions is limited to less than μ. It is therefore better not to set an upper bound on the random numbers if the value of μ is not known. However, the upper bound can be set to n in 0–1 IPPs because μ is known to be mn.

4.2 Experimental results

The experimental results of fifty 0–1 IPPs are shown in the present subsection, comparing with those of a simple CCM and a branch-and-bound method. The value of m is fixed to 10, that of a_{ij} is between 0 and 10^5, and the problems are generated randomly for each value of n, 10, 20, 30 and 40. These restrictions are set so as to make experiment easier, but they are not essential. The value of c_j is 25000 when n = 10 and it is 50000 when n = 20 for all j.

Table 1. The solution optimality of fifty 0–1 IPPs by CCM-based methods

n	CCM*			Original CCM			CCM with SA		
	Optimum probability	Average ratio	Worst ratio	Optimum probability	Average ratio	Worst ratio	Optimum probability	Average ratio	Worst ratio
10	0.97	0.998	0.83	0.012	0.62	0.14	0.42	0.95	0.77
20	0.71	0.995	0.91	0.000	0.63	0.19	0.006	0.80	0.37

The result of solving each problem eight to sixteen times by CCM* is shown in **Figure 5**.[1] The probability that the optimum solution is found by one trial is called the (one-trial) *optimum probability* in the present paper. The vertical coordinate of the left end of each polygonal line indicates the optimum probability. The optimum probability is 0.97 when n = 10, and it is more than 0.37 for all the values of n. The probability that the best solution in k trials is optimum is called the *k-trial optimum probability*. Multiple trials are measured because different solutions are found in different trials because of random numbers. Two- to sixteen-trial optimum probability and the CPU time needed for calculating the solutions are also shown in Figure 8. The eight-trial optimum probability is more than 0.89. Part of the results of the optimum probability and the ratio of the approximate and exact values of the solutions in average (CCM*-Average ratio) and in the worse case (CCM*-Worst ratio) are shown in **Table 1**.

The CCM* has been compared with three other

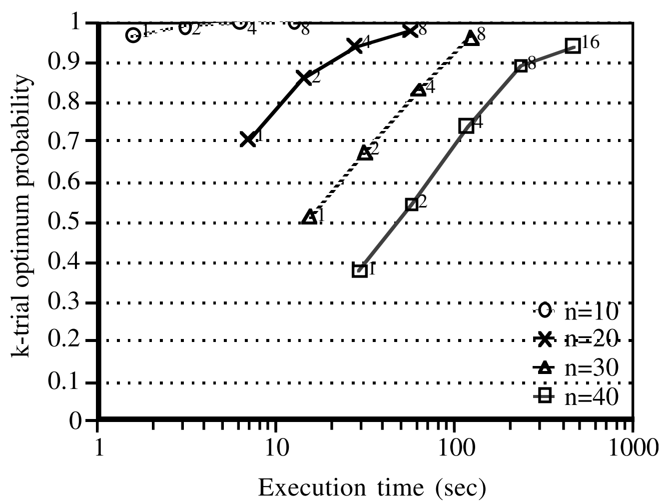

Figure 5. The optimality and CPU time of solutions of 0–1 IPPs by CCM*

[1] The performance was measured using a SOOC language processor on a Macintosh Quadra 700. The optimum solutions

methods on the same 0-1 IPP. The first method is the original CCM, and the second method is an improved version of CCM, in which a simple simulated annealing (SA) is introduced. No tunneling method is applied in these methods. The performances of these two methods are also shown in Table 1. This table shows that CCM* is much better (the optimum probability is 2.3 to 81 times higher) than the other two methods. The third method is a branch-and-bound method. Because it was not possible to measure the execution time of every problem because of CPU time limitations, the first four problems for each value of n are measured. The results are shown in **Table 2**.[1] We can probably conclude that the method based on CCM* is faster if the multiple-trial optimum probability is allowed to be 0.9 to 0.99, although a satisfactory comparison is not possible by this result, because only a small part of the problem is measured and the hardware and software are not the same. However, solutions found by the method based on CCM* are not assured to be optimum even when the computation is repeated so many times.

Table 2. The execution time (sec) of the first four 0–1 IPPs by branch-and-bound method

n	Problem 1	Problem 2	Problem 3	Problem 4	Average of 1 to 4
10	12 sec	15 sec	12 sec	15 sec	14 sec
20	73	200	20	212	126
30	172	108	630	460	343
40	487	2081	780	705	1013

5. Conclusion

Methods of solving combinatorial problems using CCM* are explained in the present paper. One of these methods, using a tunneling technique, can find a solution probably in less time in average than branch-and-bound methods in randomly generated 0–1 IPPs, if expected multiple-trial optimum probability is allowed to be 0.9 to 0.99. The tunneling method described in Section 4.1 can easily be built-in to the reaction rule compiler, and this increases the probability of discovering approximate solutions of 0–1 IPPs. CCM* can also be applied to other combinatorial problems.

Possible future work is as follows. Firstly, the relation between the distribution of upper bounds and the efficiency remains unknown. However, there is a chance of performance improvement. Secondly, a more powerful method is probably necessary for finding an optimum or nearly optimum solution in the case of more difficult problems such as the traveling salesperson problem, because there are two or more ways of composing reaction rules and their performances are different. Thirdly and most importantly, application of CCM* to dynamically changing problems in open and complex systems is the major target of this research. The current CCM* can already be applied to such problems. Other appropriate problems should be discovered and challenged using CCM*.

for the comparison were found using a branch-and-bound method.

[1] The solver of Microsoft Excel on Macintosh Quadra 840 AV has been used for this measurement. Each time in this table includes time for "formulation."

References

[For 81] Forgy, C. L.: *OPS5 User's Manual*, Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. of Computer Science, 1981.

[For 91] Forrest, S., ed.: *Emergent Computation*, MIT Press, 1991.

[Kan 92] Kanada, Y.: Toward Self-organization by Computers, *33rd Programming Symposium*, Information Processing Society of Japan, 1992 (in Japanese).

[Kan 94] Kanada, Y., and Hirokawa, M.: Stochastic Problem Solving by Local Computation based on Self-organization Paradigm, *27th Hawaii Int. Conf. on System Sciences*, 1994.

[Kan 95] Kanada, Y.: Fuzzy Constraint Satisfaction Using CCM — A Local Information Based Computation Model, *FUZZ-IEEE/IFES '95*, 2319–2326, 1995.

[Lev 85] Levy, A. V., and Montalvo, A.: The Tunneling Algorithm for the Global Minimization of Functions, *SIAM J. Sci. Stat. Comput.*, Vol. 6, No. 1, pp. 15–29, 1985.

[Shi 93] Shima, T.: Global Optimization by Annealing Type of Random Tunneling Algorithm, *Technical Reports of the Society of Instrument and Control Engineers (SICE)*, Vol. 29, No. 11, 1342–1351, 1993 (in Japanese).

[Yao 89] Yao, Y.: Dynamic Tunneling Algorithm for Global Optimization, *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. 19, No. 5, pp. 1222–1230, 1989.

[Tak 92] Takefuji, Y.: *Neural Network Parallel Processing*, Kluwer Academic Publishers, 1992.