Monte-Carlo Kakuro

Tristan Cazenave

LAMSADE, Université Paris-Dauphine,
Place Maréchal de Lattre de Tassigny, 75775 Paris Cedex 16, France
cazenave@lamsade.dauphine.fr

Abstract. Kakuro consists in filling a grid with integers that sum up to predefined values. Sums are predefined for each row and column and all integers have to be different in the same row or column. Kakuro can be modeled as a constraint satisfaction problem. Monte-Carlo methods can improve on traditional search methods for Kakuro. We show that a Nested Monte-Carlo Search at level 2 gives good results. This is the first time a nested search of level 2 gives good results for a Constraint Satisfaction problem.

1 Introduction

Kakuro, also known as Cross Sums is a popular NP-complete puzzle [5]. It consists of a predefined grid containing black and white cells. Each white cell has to be filled with an integer between 1 and 9. All cells in the same row and all cells in the same column have to contain different integers. The sum of the integers of a row has to match a predefined number, as well as the sum of the integers of a column. Table 1 give an example of a 5x5 Kakuro problem. The sum of the integers of the first row has to be 18, the sum of the integers of the last column has to be 24.

	24	25	20	26	24
18					
26					
28					
26					
21					

Table 1. A 5x5 Kakuro puzzle

Table 2 gives a solution to the problem of table 1.

The second section details search algorithms for Kakuro, the third section presents experimental results.

	24	25	20	26	24
18	1	7	5	3	2
26	4	5	3	8	6
28	5	6	7	2	8
26	8	4	1	6	7
21	6	3	4	7	1

Table 2. A solution to the previous Kakuro puzzle

2 Search Algorithms

The search algorithms we have tested are Forward Checking, Iterative Sampling, Meta Monte-Carlo search and Nested Monte-Carlo Search which are presented in this order in this section.

2.1 Forward Checking

The Forward Checking algorithm consists in reducing the set of possible values of the free variables after each assignment of a value to a given variable. It reduces the domains of the free variables that appear in the same constraints as the assigned variable.

In Kakuro, each time a value is assigned to a variable, the value is removed from the domain of the free variables that are either in the same row or in the same column as the assigned variable.

Moreover, the sum S_{row} of all the assigned variables in a row is computed, then the maximum possible value $Maxval$ for any variable in the row is computed by subtracting S_{row} to the goal sum of the row. All value that are greater than $Maxval$ are removed from the domains of the free variables of the row. If all the variables of the row have been assigned the sum is compared to the target sum and if it is different, the assignment is declared inconsistent.

A similar domain reduction and consistency check is performed for the free variables in the column of the assigned variable.

Much more elaborate consistency checks could be performed and would improve all the algorithms presented in this paper. However, our point in this paper is not about elaborate consistency checks but rather about the interest of nested search.

A Forward Checking search is a depth first search that chooses a variable at each node, tries all the values in the domain of this variable and recursively calls itself until a domain is empty or a solution is found.

The pseudo code for Forward Checking is:

```
bool ForwardChecking ()
  if no free variable then
    return true
  choose a free variable var
  for all values in the domain of var
    assign value to var
```

```
      update the domains of the free variables
      if no domain is empty or inconsistent then
        if (ForwardChecking ()) then
          return true
    return false
```

2.2 Iterative Sampling

Iterative Sampling uses Forward Checking to update the possible values of free variables. A sample consists in choosing a variable, assigning a possible value to it, updating the domains of the other free variables and looping until a solution is found or a variable with an empty domain is found. Iterative sampling performs samples until a solution is found or the allocated time for the search is elapsed.

The sample function that we give returns the number of free variables that are left when a variable has an empty domain because this value is used as the score of a sample by other algorithms.

The pseudo code for sampling is:

```
  int sample ()
    while true
      choose a free variable var
      choose a value in the domain of var
      assign value to var
      update the domains of the free variables
      if a domain is empty or inconsistent then
        return 1 + number of free variables
      if no free variable then
        return 0
```

The Iterative Sampling algorithm simply consists in repeatedly calling sample:

```
  bool iterativeSampling ()
    while time left
      if sample () equals 0 then
        return true
    return false
```

2.3 Meta Monte-Carlo Search

Rollouts were successfully used by Tesauro and Galperin to improve their Backgammon program [6], they consist in playing games according to an algorithm that chooses the moves to play. Then the scores of the games are used to choose a move instead of directly using the base algorithm. A related algorithm that has multiple levels is Reflexive Monte-Carlo search [2] which has been used to find long sequences at Morpion Solitaire. Reflexive Monte-Carlo search consists in playing random playouts at the base level, and to play a few games at the lower level of a search in order to find the best move at the current level of the search. At Morpion Solitaire, games at the meta level give better results than games at the lower level.

A Meta Monte-Carlo Search tries all possible assignments of the variable, plays a sample after each assignment and choose the value that has the best sample score. The algorithm memorizes the best sample so as to follow it in subsequent moves if no better sample has been found.

The pseudo code of Meta Monte-Carlo Search is:

```
1  int metaMonteCarlo ()
2    best score = number of free variables
3    while true
4      choose a free variable var
5      for all values in the domain of var
6        assign value to var
7        update the domains of the free variables
8        if a domain is empty or inconsistent then
9          score = 1 + number of free variables
10       else
11         score = sample ()
12       if score < best score then
13         best score = score
14         best sequence = {{var,value},sample sequence}
15     var = pop variable of the best sequence
16     value = pop value of the best sequence
17     assign value to var
18     update the domains of the free variables
19     if a domain is empty or inconsistent then
20       return 1 + number of free variables
21     if no free variable or best score equals 0 then
22       return 0
```

At line 14, when a sample has found a new best sequence, it is memorized. A sequence consists of an ordered list of variables and values that have been chosen during the sample. It is analogous to a sequence of moves in a game.

The algorithm can be used with any maximum allocated time, repeatedly calling it until a solution is found or the time is elapsed, as for Iterative Sampling:

```
bool iterativeMetaMonteCarlo ()
  while time left
    if metaMonteCarlo () equals 0 then
      return true
  return false
```

2.4 Nested Monte-Carlo Search

Nested Monte-Carlo Search [3] pushes further the meta Monte-Carlo approach, using multiple meta-levels of nested Monte-Carlo searches. This approach is similar to previous approaches that attempt to improve an heuristic of a solitaire card game with nested calls [7, 1]. These algorithms use a base heuristic which is improved with nested calls, whereas Nested Monte-Carlo Search uses random moves at the base level instead. Nested Monte-Carlo Search is an algorithm that uses no domain specific knowledge and which is widely applicable. However adding domain specific knowledge will probably improve it, for example at Kakuro pruning more values using stronger consistency checks would certainly improve both the Forward Checking algorithm and the Nested Monte-Carlo search algorithm.

The application of Nested Monte-Carlo Search to Constraint Satisfaction is:

```
1  int nested (level)
2    best score = number of free variables
3    while true
4      choose a free variable var
5      for all values in the domain of var
6        assign value to var
7        update the domains of the free variables
8        if a domain is empty or inconsistent then
9          score = 1 + number of free variables
10       else if level is 1
11         score = sample ()
12       else
13         score = nested (level - 1)
14       if score < best score then
15         best score = score
16         best sequence = {{var,value},level-1 sequence}
17     var = pop variable of the best sequence
18     value = pop value of the best sequence
19     assign value to var
20     update the domains of the free variables
21     if a domain is empty  or inconsistent then
22       return 1 + number of free variables
23     if no free variable or best score equals 0 then
24       return 0
```

Meta Monte-Carlo search is a special case of Nested Monte-Carlo Search at level 1.

It is important to memorize the best sequence of moves in Nested Monte-Carlo Search. It is done at line 16 of the nested function. At this line, if the search of the underlying level has found a new best sequence, this sequence is copied as the best sequence of the current level.

Nested Monte-Carlo Search parallelizes very well, for Morpion Solitaire disjoint version, speedups of 56 for 64 processors were obtained [4].

The algorithm can be used with any allocated time, repeatedly calling it at a given level.

2.5 Choosing a variable

When choosing a variable, the usual principle is to choose the variable that will enable to find that there is no solution under the node as fast as possible. A common, general and efficient heuristic is to choose the variable that has the smallest domain size. This is the heuristic we have used for all the algorithms presented in this paper.

2.6 Choosing a value

When choosing a value, the usual principle is to choose the value that has the most chances of finding a solution because if there is no solution, all values have to be tried in order to prove that there is no solution. In this paper we choose values at random among possible values of a variable. However, Nested Monte-Carlo Search mainly consists in getting much information about the interestingness of all values before choosing one, so it can also be considered as an algorithm that carefully selects values to be tried.

3 Experimental results

In this section we explain how problems have been generated. We then give the results of running the algorithm on various problems. We also evaluate the influence of the number of possible values on problem hardness.

3.1 Problem generation

In order to generate a problem, a search is used to generate a complete grid of a given size with the constraint that all values are different in the same column or row. Then the sum of each row and each column is computed. Then values are randomly removed until the desired percentage of holes is reached. For each percentage of holes, 100 problems have been generated.

3.2 Comparison of algorithms

The four algorithms that were tested are Forward Checking, Iterative Sampling, Nested Monte-Carlo Search at level 1 and level 2.

In order to estimate problems difficulties these four algorithms were tested on all percentages of holes of 10x10 grids with values ranging from 1 to 11. Figure 1 gives the number of problem solved for each percentage and each algorithm using a timeout of 10 seconds per problem. Figure 2 give the total time used by each algorithm for the same problems. It is clear from these figures that Nested Monte-Carlo search at level 2 easily solves almost all the problems in less than 10 seconds when Forward Checking and Iterative Sampling solve almost no problem within 10 seconds when the problems have more than 80% of free variables.

We can see that problem difficulty increase with the number of holes, the most difficult problems being the empty grids problems. This is different from a closely related problem, the quasi group completion problem also consists in filling a grid with

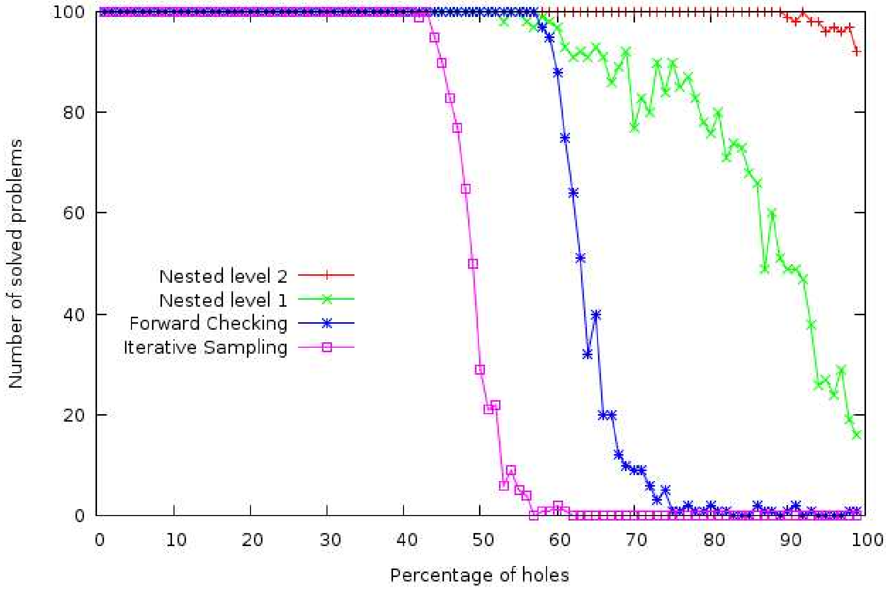

Fig. 1. Number of solved problems for different percentage of holes and different algorithms with a timeout of 10 seconds per problem, 10x10 grids, values ranging from 1 to 11

different values on each row and each column, but there is no constraints on the sum of the values and there are as many values as the size of the column or the row. The quasi group completion problem is easy for low and high percentages of holes and hard for intermediate percentages. In our experiments, Iterative Sampling very easily solves all quasi group completion problems, with any percentage of holes, up to size 10x10. Kakuro is harder to solve than quasi group completion for the same problem size, and the problems hardness does not have the same repartition.

Moreover Nested Monte-Carlo Search at level 2 is better than Nested Monte-Carlo Search at level 1 which is better than Forward Checking which is in turn better than Iterative Sampling.

We now compare algorithms giving them more time (1,000 seconds per problem) on empty 6x6 grids (36 free variables) since empty grids are the most difficult problems. Possible values range from 1 to 7. Table 3 gives the number of problems solved and the time to solve them for the different algorithms. We see that Nested Monte-Carlo Search is still the best algorithm, however Iterative Sampling becomes much better than Forward Checking on 6x6 empty grids.

In order to test more difficult problems we repeated the experiment for empty 8x8 grids with values ranging from 1 to 9. The results are given in table 4. Nested Monte-Carlo Search at level 2 is still the best algorithm while Iterative Sampling and Forward Checking perform badly. We can observe that most of the time of the Forward Checking

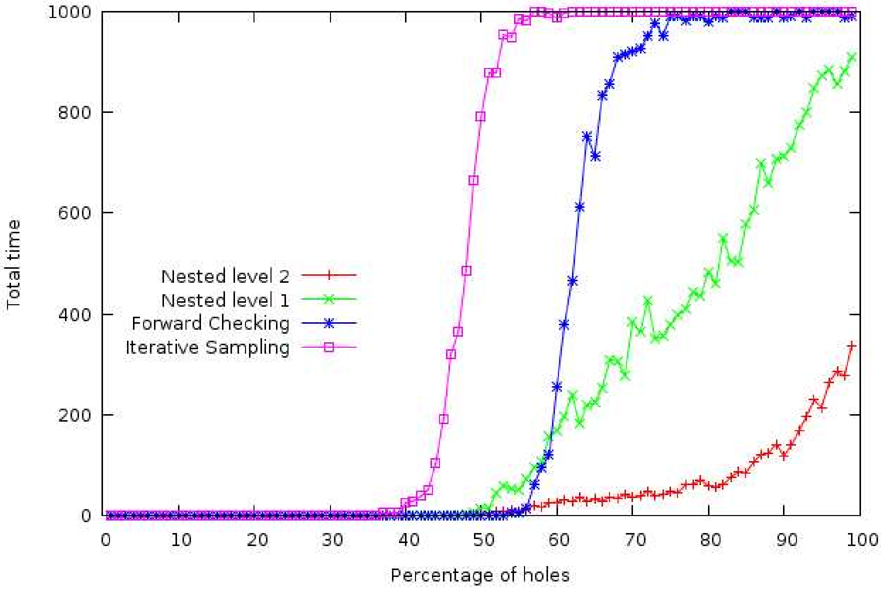

Fig. 2. Time spent for all problems for different percentage of holes and different algorithms with a timeout of 10 seconds per problem, 10x10 grids, values ranging from 1 to 11

algorithm is spent on problems that are not solved (92 problems hence 92,000 seconds) while Iterative Sampling solves two more problems but takes more time.

3.3 Influence of the number of values

The next experiment consists in estimating the evolution of the difficulty of Kakuro problems with the number of possible values. We solved 100 empty 8x8 grids with Nested Monte-Carlo Search level 2 for 9, 10, 11 and 12 possible values. The results are given in table 5. We see that the difficulty starts to increase with the number of possible values and then decreases when enough values are possible.

Algorithm	Problems solved	Total time
Nested Level 2	100 out of 100	0.91 s.
Nested Level 1	100 out of 100	1.42 s.
Iterative Sampling	100 out of 100	424.21 s.
Forward Checking	98 out of 100	9,433.98 s.

Table 3. Comparison of different algorithms for empty 6x6 grids, values ranging from 1 to 7, timeout of 1,000 seconds

Algorithm	Problems solved	Total time
Nested Level 2	100 out of 100	17.85 s.
Nested Level 1	100 out of 100	78.30 s.
Iterative Sampling	10 out of 100	94,605.16 s.
Forward Checking	8 out of 100	92,131.18 s.

Table 4. Comparison of different algorithms for empty 8x8 grids, values ranging from 1 to 9, timeout of 1,000 seconds

Algorithm	Problems solved	Possible Values	Total time
Nested Level 2	100 out of 100	[1,9]	17.85 s.
Nested Level 2	100 out of 100	[1,10]	1,939.51 s.
Nested Level 2	100 out of 100	[1,11]	1,166.13 s.
Nested Level 2	100 out of 100	[1,12]	1,214.29 s.
Nested Level 2	99 out of 100	[1,13]	1,688.51 s.
Nested Level 2	100 out of 100	[1,14]	629.47 s.
Nested Level 2	100 out of 100	[1,15]	672.90 s.
Nested Level 2	100 out of 100	[1,16]	603.38 s.
Nested Level 2	100 out of 100	[1,17]	429.26 s.
Nested Level 2	100 out of 100	[1,18]	579.99 s.

Table 5. Solving empty 8x8 grids with different numbers of possible values, timeout of 1,000 seconds

4 Conclusion

We have compared Forward Checking, Iterative Sampling and Nested Monte-Carlo Search on Kakuro problems. Nested Monte-Carlo search at level 2 gives the best results. We have also shown the difficulty of Kakuro problems given their number of holes and number of possible values.

Future work include testing the algorithms with stronger consistency checks and comparing them on other problems.

Acknowledgments

This work has been supported by French National Research Agency (ANR) through COSINUS program (project EXPLO-RA ANR-08-COSI-004)

References

1. R. Bjarnason, P. Tadepalli, and A. Fern. Searching solitaire in real time. *ICGA Journal*, 30(3):131–142, 2007.
2. T. Cazenave. Reflexive Monte-Carlo search. In *Computer Games Workshop*, pages 165–173, Amsterdam, The Netherlands, 2007.

3. T. Cazenave. Nested Monte-Carlo search. In *IJCAI 2009*, Pasadena, USA, 2009.
4. T. Cazenave and N. Jouandeau. Parallel nested Monte-Carlo search. In *NIDISC Workshop*, Rome, Italy, 2009.
5. G. Kendall, A. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.
6. G. Tesauro and G. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems 9*, pages 1068–1074, Cambridge, MA, 1996. MIT Press.
7. X. Yan, P. Diaconis, P. Rusmevichientong, and B. Van Roy. Solitaire: Man versus machine. In *Advances in Neural Information Processing Systems 17*, pages 1553–1560, Cambridge, MA, 2005. MIT Press.