


13,070,496 members (22,574 online)



home

articles

quick answers


discussions

features

community

help

Search for articles, questions, tips



Articles » General Programming » Algorithms & Recipes » Algorithms



Article

Browse Code

Stats

Revisions

Alternatives

Comments (16)

Add your own alternative version

Tagged as

VC7

C++

Windows

Visual-Studio

STL

Dev

Intermediate

Stats

138.7K views

4.4K downloads

42 bookmarked

Posted 7 Jun 2004

CPOL

# A brute force search algo

**Giannakakis Kostas**, 7 Jun 2004  
★★★★☆ 3.94 (14 votes)    Rate this:

A generic class implementing an exhaustive searching algorithm for solving a variety of puzzles and riddles

- Download source - 15 Kb
- Download demo application - 86 Kb
- Download java applets source - 67 Kb

## Introduction

One technique to solve difficult problems using a computational system, is to apply brute force search. This means to exhaustively search through all possible combinations until a solution is found. In this article I will present an implementation of a brute force search algorithm, that can be applied to a variety of problems. The idea for this article was taken by an [article](#) in CodeProject, which presented a C++ program that solves Einstein's Riddle. The functionality of this article was extended, in order to exploit the benefits of Object Oriented Programming and to produce a generic solver class, that can be used as base class from many problem-specific solver classes. Many examples of problem-solver classes will be presented, including one that solves Einstein's Riddle.

*Note:* The language of the implementation is C++. Although some advanced features of the language are used (template classes, STL), the technical details in this article are kept in a minimum, so that even the less-experienced programmer can follow the logic of the brute force algorithm and have fun with the example puzzles.

## Eight Queens Riddle

I will introduce the idea of the brute force search algorithm starting from the Eight Queens Riddle. This is the well-known problem of placing 8 non-attacking queens on a chessboard. Source code for solving this puzzle can be found in many software books. In the download section you will find, a Java applet ([java\\_applets\\_src.zip](#)) that implements the riddle.

A computing system can help us solve the riddle quickly. Moreover it can help us find all possible solutions. To do so we must instruct our program to search all possible placements of 8 queens on the chessboard. There is a total of  $64!63!62!61!60!59!58!57!$  different placements. It would be highly inefficient to try to search all these different combinations. It is much better to solve the problem inductively. That is to first solve the problem of placing one queen at a 1x8 chessboard, then the problem of placing two queens at a 2x8 chessboard and continue until the full solution is found. In this way large groups of apparently erroneous combinations are caught early in the process and need not be tested. If there is a pair of attacking queens at the first n columns of the chessboard, then no matter where the rest 8-n queens are placed, the combination can not constitute a solution.

The inductive searching algorithm works as follows. We start by placing the first queen at a1. The second queen is placed at b3 (as b1 and b2 are attacked by the first queen). Then we look for an unattacked square at the third column. We continue in the same manner, until we reach a column, which has no unattacked squares. In this case we return to the previous column and we try to move the queen placed there to another unattacked square. We only examine squares that are in a higher row than the previous one. If this is impossible too, we move another column back and we try to do the same. If a new unattacked square at a previous column is found, we are ready to move forwards again. Everytime we are able to place a queen at eighth row, a new solution is found. We increment a variable, store the positions of the queens in a table and we return to column 7 to find a new unattacked square. The searching is complete when we need to return from column 2 to column 1 and the queen is placed at a8. A visualisation of the algorithm can be presented by clicking "Solve" button at the previous applet.

## Generalisation

The concept of the inductive searching algorithm can be used in more general situations. To be able to re-use the functionality of the algorithm we'll define an "abstract" riddle. This "abstract" riddle is represented by a table of n cells, which must be filled by elements coming from an ordered list (see Figure 1).

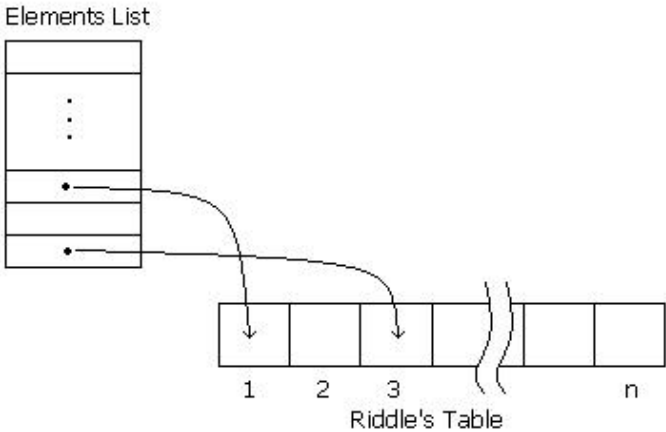


Figure 1: General Riddle

To solve the riddle inductively we first fill cell 1, then cell 2 and so on until all cells are filled. The flowchart of the algorithm is presented in the following figure:

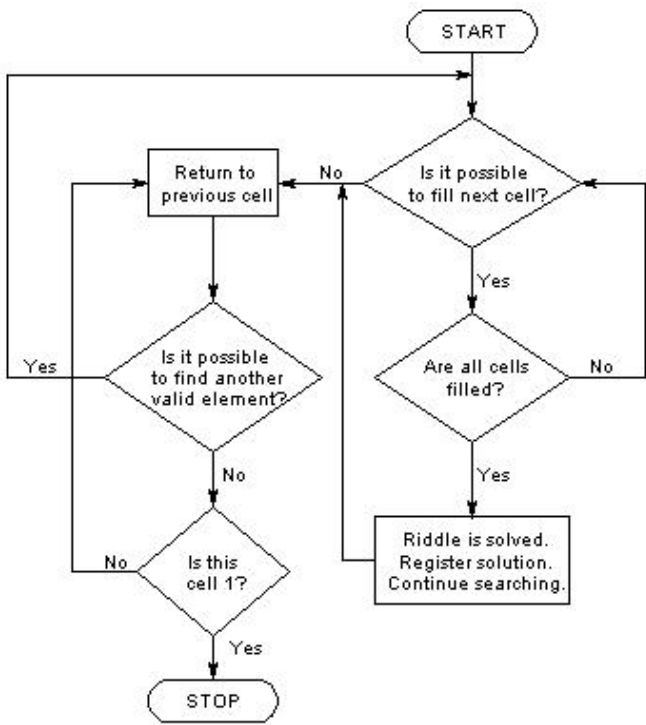


Figure 2: Flowchart

For this technique to work we must define two functions:

- `check_placement(m)` : which checks if the sub-riddle of first m cells is solved
- `get_next_element(e)` : which given an element e, returns the element immediately after it in the ordered list.

The C++ class `riddle_solver<move_t>` implements the "abstract" riddle.

Hide Shrink Copy Code

```
template <class move_t>
class riddle_solver
{
public:
    /** Constructor. */
    riddle_solver<move_t>();

    /** Destructor. */
    ~riddle_solver<move_t>();

    /**
     Solves the riddle using inductive searching algorithm.
     If not instructed otherwise
     (by restricting the number of solutions to search for),
     all possible combinations
     are searched.

     @result Returns true if one or more solutions have been found.
     */
    bool solve();

    /**
     Prints all solutions found.
     */
    void print_all();

    /**
     Sets the maximum number of solutions to search for.
     If 0 is given, solve function
     will scan all possible combinations.

     @param n Maximum number of solutions to search for.
     */
    void set_solutions_count(int n);

    /**
     Sets the size of the riddle, that is the size
     of the table it must be filled
     to solve the riddle.

     @param s Riddle's size.
     */
    void set_size(int s);

    /**
     Sets null element. When null element is given as input
     to get_next_element function,
     the first element in the ordered element list will
     be returned. Null element cannot
     be used to fill a cell.

     @param mn Null element.
     */
    void set_null_element(move_t mn);

private:
    /**
     tries to fill the next cell in the table.
     If this is impossible it will return
     false. The algorithm must then return to a previous cell.
     It must not be overridden.

     @result Returns true if it was possible to make a new move.
     */
    bool make_next_move();
```

```
/**
 * It tries to find another valid element in a previous cell.
 * It will go back as many
 * cells as required. If this impossible,
 * false will be returned and the searching
 * will stop. It must not be overridden.
 *
 * @result Returns true if a new valid element in
 * a previous cell has been found.
 */
bool go_back();

protected: // virtual functions

/**
 * Checks if the subriddle of size moves_count is solved.
 * It may be overridden. It is
 * declared virtual, because it is called from solve function
 * of the base class.
 * When called in the context of a subclass,
 * solve function must call check_placement
 * of the subclass and not this version.
 *
 * @result Returns moves_count if the subriddle is solved, 0 otherwise.
 */
virtual int check_placement();

/**
 * Gets next element from the elements ordered list.
 *
 * @param pos Position of cell that contains the
 * element to be replaced by the
 * next one found. A new cell will contain null
 * element, so that we start the
 * searching from the first element in the list.
 * It must be overridden.
 * @result Returns next element.
 */
virtual move_t get_next_element(int pos);

/**
 * Registers move at specified position.
 * In its simplest form, it just inserts
 * element move at cell pos. It may be overridden.
 *
 * @param pos Position of cell to place the element.
 * @param move Element to be placed.
 */
virtual void register_move(int pos, move_t move);

/**
 * Undoes last move (at position moves_count - 1).
 * In its simplest form, it just
 * removes the element placed at the last cell and
 * fills it with null element. It may
 * be overridden.
 */
virtual void unregister_last_move();

/**
 * Prints solution in the standard output.
 *
 * @param sol Vector containing the solution.
 */
virtual void print_solution(vector<move_t> sol) {};;

protected: // Data members

/// Table storing the moves made.
vector<move_t> moves_array;

/// Riddle's size (moves_array size).
int size;

/// Number of moves made so far.
int moves_count;

private:
/// Null element.
move_t move_null;

/// Maximum number of solutions to look for.
int sol_number_max;

/// Vector for storing all solutions found.
vector<vector<move_t> > sol_v;
};
```

Note: `riddle_solver<move_t>` is a template class. `move_t` type defines the type of the elements that fill the cells of the table. In all the examples of this article

Hide Copy Code

```
riddle_solver
```

will be instantiated with int as `move_t`. However the class can be used with any other native type (such as float or double) or user-defined class that overloads = operator.

Using `riddle_solver<int>` as base class, the eight queens riddle can be solved with a class as simple as that:

Hide Shrink Copy Code

```
q8_riddle_solver::q8_riddle_solver()
```

```
{
    set_null_element(-1);
    set_size(8);
}

q8_riddle_solver::~q8_riddle_solver() {}

int
q8_riddle_solver::get_next_element(int pos)
{
    if (moves_array[pos] == 7)
        return(-1);
    else
        return(moves_array[pos]+1);
    return(-1);
}

int
q8_riddle_solver::check_placement()
{
    int i, j;

    for(i=0;i<moves_count;i++)
    {
        for(j=i+1;j<moves_count;j++)
        {
            if (moves_array[i] == moves_array[j])
                return(0);
            if (i + moves_array[i] == j + moves_array[j])
                return(0);
            if (i - moves_array[i] == j - moves_array[j])
                return(0);
        }
    }

    return(moves_count);
}

void
q8_riddle_solver::print_solution(vector<int> sol)
{
    char str[4];
    str[2] = ' ';
    str[3] = '\0';

    for(char i = 0; i < 8; i++)
    {
        str[0] = 'a' + i;
        str[1] = '1' + sol[i];
        cout << str;
    }
    cout << endl;
};
```

In the following paragraphs **riddle\_solver** class will be used for solving more puzzles.

## Einstein's Riddle

This is the riddle that originally gave me the motivation for this article. This is how it goes:

- There are 5 houses in five different colors.
- In each house lives a person with a different nationality.
- These five owners drink a certain type of beverage, smoke a certain brand of cigar and keep a certain pet.
- No owners have the same pet, smoke the same brand of cigar or drink the same beverage.

For this bizarre neighbourhood we know the following:

1. the Brit lives in the red house.
2. the Swede keeps dogs as pets.
3. the Dane drinks tea.
4. the green house is on the left of the white house.
5. the green house's owner drinks coffee.
6. the person who smokes Pall Mall rears birds.
7. the owner of the yellow house smokes Dunhill.
8. the man living in the center house drinks milk.
9. the Norwegian lives in the first house.
10. the man who smokes blends lives next to the one who keeps cats.
11. the man who keeps horses lives next to the man who smokes Dunhill.
12. the owner who smokes BlueMaster drinks beer.
13. the German smokes Prince.
14. the Norwegian lives next to the blue house.
15. the man who smokes blends has a neighbour who drinks water.

The question is: who owns the fish? To answer this question one must find the nationality, colour, pet, beverage and smoke that correspond to all the houses.

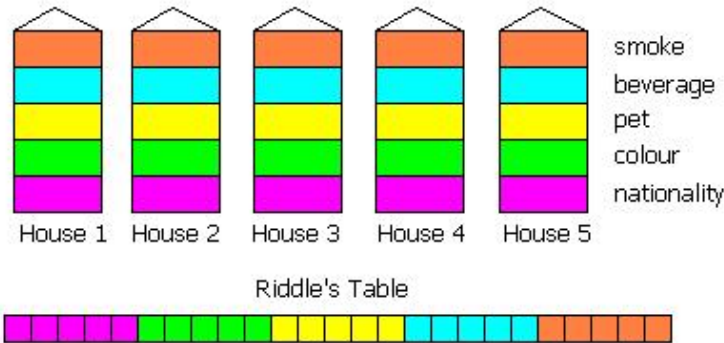


Figure 3: Einstein's riddle neighbourhood

To adapt the riddle to the form of our abstract riddle, we define a table of size 25. First 5 cells contain the nationality of the owners, the next five the colour of the houses, then the pets the beverages and the smokes. This means that while we construct a solution, we will first find the nationalities of all owners, then move to the colours of the houses and so on. The elements list consists of numbers 1 to 5, with the obvious ordering. We have 5 property types (nationality, colour, pet, beverage and smoke) with 5 property values each. We map each property value to a number from 1 to 5. The cells of the table are filled with numbers from 1 to 5, however there is the restriction that two houses can't contain the same value for any given property. The details of the implementation appear in files eisteins\_riddle.cpp and einsteins\_riddle.hpp.

## Magic Squares

A magic square of order n is a nxn table, containing all numbers from 1 to n² in such a way, that the sum of all rows, columns and primary diagonals is the same. Algorithms for constructing a magic square of an arbitrary order exist [1]. There is also an [article](#) in CodeProject that will do the job. The problem of enumerating all magic squares of a specific order sounds more challenging ([2]). We will try to use `riddle_solver` class to enumerate the magic squares of order 3, 4 and 5.

In the concept of the abstract riddle presented above, we are going to construct the magic square cell by cell, visiting each one of them with the order presented below.

1	2	3
4	5	6
7	8	9

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 4: Visiting order for magic squares

The following listing presents the implementation of

Hide Copy Code

check\_placement

routine. Whenever a cell is filled with a number, the routine checks if a row, a column or a primary diagonal is completed. If this has happened, it calculates the sum and tests if it is the proper one. The sum of all rows, columns and primary diagonals of a magic square nxn should be  $n[n^2+1]/2$ .

Hide Shrink Copy Code

```
int
magic_square_solver::check_placement()
{
    int r, c;

    if (moves_count == 0)
        return(0);

    r = (moves_count-1) / square_size;
    c = (moves_count-1) % square_size;

    // A row has just been completed
    if (c == square_size-1)
    {
        if (!test_row(r))
            return(0);
    }

    if (r == square_size - 1)
    {
        // Reverse diagonal has been completed
        if (c == 0)
        {
            if (!test_reverse_diagonal())
                return(0);
        }
        // Diagonal has been completed
        if (c == square_size - 1)
        {
            if (!test_diagonal())
                return(0);
        }

        // A column has just been completed
        if (!test_column(c))
            return(0);
    }

    return(moves_count);
}
```

Although this code works well for 3x3 squares, you will notice that it takes way too long to be of any use for 4x4 squares. To make it faster we have to enhance `check_placement` routine in order to detect erroneous combinations earlier. We achieve this by modifying `check_placement`, so that it makes tests for every new cell filled, not just for the cells that complete a row, a column or a primary diagonal. For every newly filled cell we calculate the sums of the partially filled row and column it belongs to. If it is a part of primary diagonal, we also calculate the sum of the diagonal so far. Then we check, if any of these partial sums is already bigger from the desired sum or if it is less than the minimum it should be. If this is the case, we return one cell back. The minimum sum for a partially filled row/column/diagonal is calculated as follows. If n-m cells are filled, the partial sum should be equal or greater of  $n[n^2+1]/2 - n^2 - (n^2-1) - \dots - (n^2-m+1)$ .

Hide Shrink Copy Code

```
int
magic_square_solver::check_placement()
{
    int r, c;

    if (moves_count == 0)
```

```
        return(0);

    r = (moves_count-1) / square_size;
    c = (moves_count-1) % square_size;

    // A row has just been completed
    if (c == square_size-1)
    {
        if (!test_row(r))
            return(0);
    }

    if (r == square_size - 1)
    {
        // Reverse diagonal has been completed
        if (c == 0)
        {
            if (!test_reverse_diagonal())
                return(0);
        }
        // Diagonal has been completed
        if (c == square_size - 1)
        {
            if (!test_diagonal())
                return(0);
        }

        // A column has just been completed
        if (!test_column(c))
            return(0);
    }
    else
    {
        // Test partial row, column, diagonal
        if (c != square_size-1 && !test_partial_row(r, c))
            return(0);
        if (!test_partial_column(c, r))
            return(0);
        if (c == r && !test_partial_diagonal(r))
            return(0);
        if ((c == square_size - r - 1) && !test_partial_reverse_diagonal(r))
            return(0);
    }

    return(moves_count);
}
```

With the enhanced **check\_placement** routine you will be able to enumerate the 4x4 squares in a couple of minutes. Compare the result you will find with that presented in [2]. Note that if we have a solution, we can easily find 7 more by means of reflections and rotations. If these variations of the same solution aren't counted, we must divide our result with 8.

When we move to 5x5 squares, we notice that even the enhanced version is inefficient. Actually there are 275305224\*8 magic squares of 5-order. If the algorithm were able to find a solution every 1ms, it would still require 25 and half days to enumerate all the solutions! No need to get discouraged though. With a simple trick we can boost the speed of the algorithm. All we have to do is to change the order we visit the cells:

1	2	3	4	5
6	10	11	12	13
7	14	17	18	19
8	15	20	22	23
9	16	21	24	25

Figure 5: Visiting order for 5x5 magic squares

The **magic\_square\_5** class solves the problem of 5x5 magic squares following the visiting order of the above figure. The class performs only basic tests, when a row, a cell or a primary diagonal is completed. In my machine it was able to find about one or two solutions every second. It is still very slow but I believe that with a few optimisations it could be tweaked to enumerate all magic squares of order 5 in a decent period of time.

From this example one can conclude that is very important to catch invalid combinations as early as possible, even if that means that **check\_placement** routine must be made more complicated and thus more time consuming. The benefit from limiting the number of combinations that must be tested could be of much greater importance than the time spent in **check\_placement**. The same effect (limiting the number of combinations) can be achieved by changing the order of filling the table.

## Peg Solitaire

This is a classic board game. To win the game you must remove all the pegs off the board, except of the last one. A peg can move to a free position, two positions at the left, right, up or down, jumping over another peg. The peg that was jumped over is removed. You can play the game with a Java applet you will find in the download section.

To solve this riddle using **riddle\_solver** class, the riddle's table and the elements list must be defined. The riddle is solved when 31 moves are made, therefore the riddle's table should be of size 31 and contain all the moves required to win the game. We must also find a way to represent a move. This is done based on the position of the peg that is moving and the direction of the move (see Figure 4).



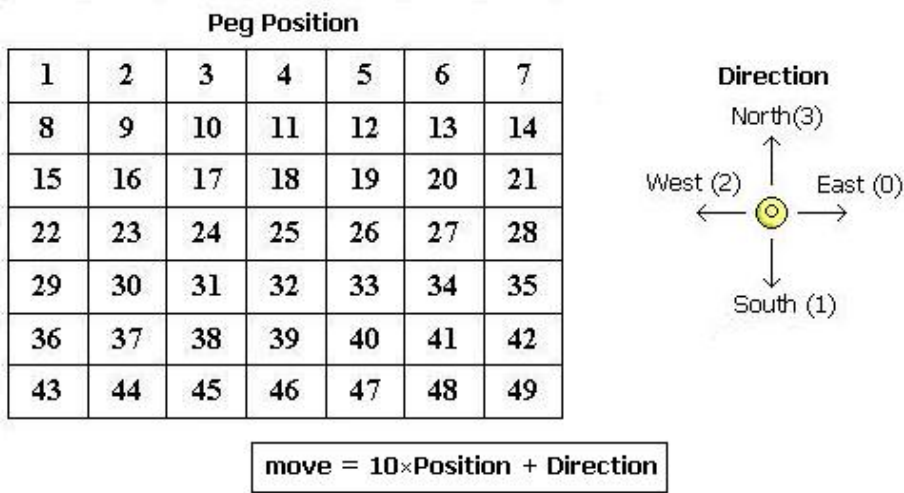


Figure 6: Representation of moves

The moves are ordered based on their numeric values, that is a move with a higher numerical value is tried after a move with a lower numerical value. The details of the implementation can be found in solo.cpp and solo.hpp files.

## Links

- [MathWorld](#) - Information about magic squares from MathWorld.
- [Enumeration of magic squares](#) - A page about enumeration of magic squares.
- [My homepage](#) - Java puzzles.

## History

- 3 June 2004. Initial release

## License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

## Share

EMAIL

TWITTER

## About the Author



### Giannakakis Kostas

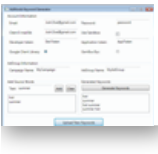
Software Developer (Senior) Self employed  
Greece

No Biography provided

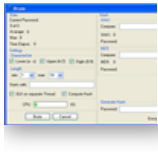
## You may also be interested in...



[HackerSpray - Block Brute force and DOS attacks](#)



[Generate and add keyword variations using AdWords API](#)



[Brute Force](#)



[Window Tabs \(WndTabs\) Add-In for DevStudio](#)



[SAPrefs - Netscape-like Preferences Dialog](#)



[OLE DB - First steps](#)

## Comments and Discussions

You must [Sign In](#) to use this message board.

Search Comments

Go

Spacing

Relaxed

Layout






















Thread View

Per page

25

Update

FirstPrevNext


	<a href="#">Great article</a> 	 Albert_Nik	27-Jul-12 0:21	2
	<a href="#">Great Code!!</a> 	 Diya Burman	24-May-12 4:20	2
	<a href="#">Thank You.The source code which i waited for so long...</a> 	 bnc	9-Nov-08 4:40	2
	<a href="#">Geia sou Elladara</a> 	 extus	10-Oct-04 11:07	4
	<a href="#">I liked it very much</a> 	 Jim Xochellis	18-Jun-04 20:02	3
	<a href="#">alpha beta?</a> 	 Vladimir Ralev	9-Jun-04 6:22	2
	<a href="#">Good search algo,but</a> 	 Randy Li	9-Jun-04 4:10	1


Last Visit: 31-Dec-99 18:00


Last Update: 6-Aug-17 15:14


[Refresh](#)


1


 General


 News


 Suggestion


 Question


 Bug

 Answer

 Joke

 Praise

 Rant

 Admin