Case 2:
Examples of systematic search

Antti Valmari
Tampere University of Technology

File \LaTeX ed September 17, 2009
1 On the use of systematic search

1.1 Systematic search

Systematic search means trying every possible candidate for a solution.

The number of candidates is typically exponential in the size of the input:

- e.g., $2^{\text{size}}$
- e.g., $n$ cities can be visited in $n! \geq (n/2)^{n/2}$ different orders
  - $1 \cdot 2 \cdot 3 \cdot 4$, $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$
  - $1 \cdot 2 \cdots \lceil n/2 \rceil \cdot \lfloor n/2 + 1 \rfloor \cdots n$
  - $(n-2)!$, if the first and last city have been fixed

$\Rightarrow$ Only possible if the size is small

- 15 cities $\Rightarrow 1,3 \cdot 10^{12}$ orders
- 30 cities $\Rightarrow 2,7 \cdot 10^{32}$ orders
- there have been $4,3 \cdot 10^{29}$ picoseconds after the Big Bang
Often a solution candidate can be built one part at a time

Often it is possible to see of an incomplete candidate that it cannot be completed to a solution

- e.g., when searching for a shortest path, the incomplete candidate is already longer than the shortest complete path found so far

⇒ by rejecting it one rejects simultaneously all complete candidates to which it can be completed

⇒ a significant amount of work is saved

Usually the amount of work will still remain exponential

⇒ We can make the size of problems that can be solved in practice grow, but not without limit
1.2   When is it worth using?

Systematic search is worth using only if

- the input is very small, or
- better algorithms that solve the problem are not found

There are strong reasons to believe that there cannot be essentially better general algorithms than systematic search for a significant set of problems

- the theory of so-called \textit{NP-completeness}
  - the topic of another set of lectures
- at least it is certain that it is not wise to trust on finding such an algorithm when needed!

Even if a good algorithm has been invented, finding it from the literature may be too difficult

- finding the right keywords for a web or literature search is difficult
- recognizing that a given algorithm is suitable requires insight
- the literature is hard to read
- some algorithms work well only in theory
  - significant improvement in efficiency is only obtained with unrealistically big inputs
- a tiny difference in the statement of the problem may prevent the use of an algorithm
- an algorithm can be really complicated
  ⇒ Finding a better algorithm than systematic search from the literature requires special expertise in algorithms
Designing an algorithm on one’s own requires even more competence
  ⇒ One must sometimes use systematic search
In some cases a well-implemented systematic search algorithm provides perfectly acceptable performance on modern computers
- e.g., 9 × 9 sudoku
If finding a better algorithm is important enough, one may, of course, consult an expert
1.3 On implementing systematic search

In systematic search, certain basic operations are repeated again and again

- adding one component to a solution candidate
- testing the validity of a candidate
- removing one component from a candidate

⇒ Worth implementing efficiently

⇒ It is worth the effort to put elementary algorithmic tricks into good use

- systematic search brings differences in basic level algorithms clearly forward

⇒ are good to illustrate the topics of this course even if mastering systematic search
  is not considered as an important goal in itself

The order in which solution candidates are built can have a tremendous effect

- in particular the order in which new components are added to the candidate
  - e.g., is an incomplete path continuous, or does it consist of separate segments
• also the order in which mutually exclusive alternatives to the same place in the candidate are tried
  – e.g., what neighbour city of the last city of an incomplete continuous path is chosen as the next city

⇒ Worth spending some thought on how to choose

Testing the validity of a candidate often involves making a compromise

• a quick coarse test or a slow more precise test?

⇒ many candidates that can be tested quickly or fewer candidates whose testing is slower?

• testing can perhaps be made faster with suitable data structures, but then it must be possible to update them efficiently as the candidate evolves

Again thinking is needed, and perhaps testing

Altogether this field offers interesting challenges, to which one may apply skills at many levels!
2 The $n \times n$ queens’ problem

2.1 Statement of the problem

One must put $n$ queens on a $n \times n$ chessboard so that they do not threaten each other:

- not two queens on the same row
- ... column
- ... ascending or descending diagonal

This is a classical example of systematic search.
2.2 The basic principle of searching a solution

8 queens can be put on a $8 \times 8$ chessboard in altogether

\[ \frac{64!}{56!8!} \approx 4,4 \times 10^9 \]
different ways

- trying them all would be too much work

A solution can be built stepwise by putting one queen at a time on the board and checking whether it threatens the queens already on the board

There is precisely one queen on each column in a complete solution

⇒ Let step $i$ in building a solution be the putting of a queen on column $i$

- only $n$ alternative positions

The positions can be tried from top to bottom, for instance
When building a solution is in a dead end, the algorithm returns to the previous column and moves its queen one step downwards.

Eventually either a solution is found, or the first queen runs out of positions.

If a solution has been found, the search can be continued to find the next solution.
2.3 Program code

```cpp
int nr_solutions = 0;
const int max_size = 100;
int size = -1; // -1 tells: the size has not been given

#include <ctime>
time_t solv_time = 0;

int main(){
    while( size < 0 || size > max_size ){
        std::cout << "Give the size in 0 .. " << max_size << ": ";
        std::cin >> size; std::cout << std::endl;
    }
    build_solution();
    std::cout << nr_solutions << ' ' << solv_time << std::endl;
}
```
// rows and columns are indexed 0 ... size − 1
int column = 0, max_column;
int place[ max_size ]; // the location of a queen on a column

void build_solution(){
    solv_time = time( 0 );
    max_column = size - 1;
    place[ column ] = -1; // +1 takes to the first place
    while( column >= 0 ){ // terminate, when backtracking over the left side
        if( next_place() ){ // go to the next unthreatened position on the column
            if( column < max_column ){
                ++column; place[ column ] = -1; // move to the next column
            }else{
                ++nr_solutions; print_solution(); // solution found
            }
        }else{ --column; } // backtracking
    }
    solv_time = time( 0 ) - solv_time;
}
2.4 Finding an unthreatened place

Whether a row is free from threats can be tested like this, for instance

```c
for( il = 0; il < column; ++il ){
    if( place[ il ] == row ){ return false; }
}
return true;
```

A bad feature

- the test becomes more and more time-consuming when moving to the right
- for each putting of a queen on column $c$, $n$ positions are tested on column $c + 1$

$\Rightarrow$ more tests are done near the right side than near the left
  - to the extent the algorithm gets that far right
A more powerful solution: keep track of rows and diagonals if they are reserved

- horizontal rows very simply

  ```cpp
  bool reserved_row[ max_size ] = {};
  ...
  if( !reserved_row[ row ] && ... ){
      reserved_row[ row ] = true; // when progressing
      ...
  }
  ...
  reserved_row[ row ] = false; // when backtracking
  ```

- ascending and descending diagonals in a slightly more complicated way

  reserved_ascending[ row + column ]
  reserved_descending[ row - column + size ]

  - why not reserved_descending[ row - column + size - 1 ]?

⇒ Testing is constant time and fast in practice, and the same holds of the updating of the necessary information!
2.5 Program code

bool
reserved_row[ max_size ] = {},
reservedAscending[ 2*max_size - 1 ] = {},
reservedDescending[ 2*max_size ] = {};

inline bool next_place(){

    // Wipe out information on the old position, if exists.
    if( place[ column ] >= 0 ){
        int pc = place[ column ];
        reserved_row[ pc ] = reserved_ascending[ pc + column ] =
        reserved_descending[ pc - column + size ] = false;
    }
}
// Find the next unthreatened place.
while( true ){
    int pc = ++place[column];
    if( pc >= size ){ return false; } // ran out of places
    if( 
        !( reserved_row[ pc ] ||
            reserved_ascending[ pc + column ] ||
            reserved_descending[ pc - column + size ]
        )
    ){
        reserved_row[ pc ] = reserved_ascending[ pc + column ] =
            reserved_descending[ pc - column + size ] = true;
        return true; // found
    }
}
2.6 Observations

Running time measurements (g++ ... -O2, on a lap top of year 2001 or so)

- the solutions are printed

<table>
<thead>
<tr>
<th>queens</th>
<th>solutions</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>724</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2680</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>14200</td>
<td>28</td>
</tr>
</tbody>
</table>

- the solutions are not printed

<table>
<thead>
<tr>
<th>queens</th>
<th>solutions</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2680</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>14200</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>73712</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>365596</td>
<td>74</td>
</tr>
</tbody>
</table>

Implementing systematic search efficiently does not necessarily require a complicated program.

The program does not contain a two-dimensional array that corresponds to the chessboard!
2.7 To ponder

The horizontal or vertical mirror image of a solution is also a solution. Would it be beneficial to take advantage of this to save running time?

Near the right side most of the rows are reserved.

- Sketch a data structure with which the next free row is found quicker than by testing all rows starting from the current row.
- Would it pay off to start using the data structure?
3 The "Pulmakulma" problem

3.1 Statement of the problem

In Finland, Nelostuote Oy produces under a licence from Gabriel Industries, Inc.
A $6 \times 10$ board that must be filled with 12 pieces of 5 squares and different shapes

- all different shapes that consist of 5 connected squares
3.2 The solution building order

If the solution is built in a bad order, then tremendous amounts of time may be spent in developing solution candidates that lead to a dead end.

⇒ the program ensures in the following order that every square is filled:

A free square is filled by putting a piece such that its left side fills the square:

- the same piece also fills four other squares from above, above right, right, or below right.
3.3 Masks

A piece can fill the board in 1, 2, 4, or 8 different ways.

⇒ It makes sense to distinguish between a piece and a mask.

- A mask models the orientation of a piece.

⇒ Each piece has 1, 2, 4, or 8 masks.

The rotation of pieces need not be computed when building a solution candidate.

- Rotation is computed only once, namely when creating the masks.

The testing of symmetric orientations is avoided.
A mask can be represented, e.g., by listing the positions of the other four squares relative to the square that is being filled

- e.g., (1,0), (1,-1), (1,1), (2,0)

$$\Rightarrow \forall i: x_i \geq 0 \text{ and } \forall i: x_i = 0 \rightarrow y_i > 0$$

Preliminary code

```cpp
class mask_type{
    int mx[size_of_piece - 1], my[size_of_piece - 1];
};

const int size_of_piece = 5, number_of_pieces = 12,
number_of_masks = number_of_pieces * 8;

mask_type mask[number_of_masks];
```

- less than 8 * number_of_pieces masks would suffice
- however, finding the precise number in advance is hard
- the data structure is altogether so small that only little memory could be saved by using the precise number
3.4 Tests that are needed when putting a next mask

To put a next mask the program has to find

- the next **free square** in the above-mentioned order

- a **free mask** that **fits** in the free square

A good way to find the next **free square** is by trying squares one by one in order

- thanks to its simplicity it is fast, even if a long distance has to be scanned

- at any instant of time, only a limited number of squares may have been covered "in advance"

  ⇒ the distances that have to be scanned are short, on the average

  - the longest example I found is 9

- it is difficult to invent a reasonable method with which scanning could be avoided
We employ a 2-dimensional array to model the board

```c
/* The board */
const int x_size = 10, y_size = 6;
int the_board[ x_size ][ y_size ];
```

- 0 = free, others = reserved

A **mask** is **free**, if none of the masks of the same piece is reserved

⇒ is / is not reserved is actually a property of the piece, not the mask

⇒ Do pieces need a representation of their own?

- if it is fast to find the different masks of the same piece, they can all be marked reserved
- it is, however, simple and efficient to make a record for each piece that stores the "reserved" information, and to which there is a link from the mask record

A mask **fits in** a place, if

- the squares that it covers are free
- it does not extend over any side of the board
Regarding the covered squares,

- their places are found quickly using the above-mentioned relative positions:

\[
\text{the\_board[ xx + \ldots.mx[i1] ][ yy + \ldots.my[i1] ]}
\]

- whether they are reserved can be checked directly from the board

There are at least two efficient ways to test whether the mask extends over a side of the board:

1. by surrounding the board with initially reserved squares, so that the test whether the covered squares are reserved will catch overflows

2. a mask is augmented with information on its extents, and the sums of that with the coordinates of the original square are compared to the sides of the board

The first method requires one of the following

- reading beyond the side of the board does not cause a run-time error
- there are many (as many as 4) layers of additional initially reserved squares
- the squares of a mask have been listed in such an order that the extra layer is always tested before positions beyond the extra layer
The second method is straightforward to program, and requires slightly less computation when catches an overflow

⇒ We choose the second method

We also store into a mask the relative position of its first square (always (0,0)) for a reason that is explained later

```cpp
const int size_of_piece = 5, number_of_pieces = 12,
        number_of_masks = number_of_pieces * 8;
int pieces = 0, masks = 0;

struct mask_type{
    int mx[size_of_piece], my[size_of_piece]; // position of the square
    int width, height, depth; // max x, max y and min y
    int piece_nr; // number of the piece whose one orientation the mask represents
};
```
The following hold during the use of a mask

\[ mx[0] = my[0] = 0 \]
\[ \forall i: 0 \leq mx[i] \leq \text{width} \]
\[ \exists i: mx[i] = \text{width} \]
\[ \forall i: mx[i] = 0 \rightarrow my[i] \geq 0 \]
\[ \forall i: \text{depth} \leq my[i] \leq \text{height} \]
\[ \exists i: my[i] = \text{depth} \]
\[ \exists i: my[i] = \text{height} \]

The purpose of the additional information in the piece will become clear later

```cpp
class piece_type{
    public:
        int first_mask, number_of_masks; // the masks of the piece in mask
        int nx, ny; // the position on the board now (mask's square 0)
        int previous; // the previous mask that has been put on the board
        inline bool is_free(){ return nx == -1; } inline void release(){ nx = -1; }
};
```
piece_type piece[ number_of_pieces ];
mask_type mask[ number_of_masks ];

// Puts mask number $m1$ to $(x_1,y_1)$ if possible.

bool put_mask( int m1, int x1, int y1 ){
    mask_type &mk = mask[ m1 ];
piece_type &pc = piece[ mk.piece_nr ];

    // Is the mask (actually, the corresponding piece) reserved?
    if( !pc.is_free() ){ return false; } 

    // Does the mask extend over a side of the board?
    if(
        x1 + mk.width >= x_size ||
        y1 + mk.height >= y_size ||
        y1 + mk.depth < 0
    ){ return false; }
// Does the mask cover a reserved square?
for( int il = 0; il < size_of_piece; ++il ) {  // or il = 1
    if( the_board[ x1 + mk.mx[ il ] ][ y1 + mk.my[ il ] ] ) {
        return false;
    }
}

// Put the mask on the board and mark the piece as reserved (= store position).
for( int il = 0; il < size_of_piece; ++il ) {
    the_board[ x1 + mk.mx[ il ] ][ y1 + mk.my[ il ] ] = mk.piece_nr + 1;
}

pc.nx = x1; pc.ny = y1;
return true;
// Remove the mask from the board and mark the piece free.
void remove_mask( int m1 ){
    mask_type &mk = mask[ m1 ];
    piece_type &pc = piece[ mk.piece_nr ];
    check( !pc.is_free(), "Removing a free piece" );
    for( int il = 0; il < size_of_piece; ++il ){
        the_board[ pc.nx + mk.mx[ il ] ][ pc.ny + mk.my[ il ] ] = 0;
    }
    pc.release();
}

Putting a mask on and removing it from the board benefit a bit from the presence of the square (0,0) in the mask, but this is not the most important reason
3.5 Construction of the pieces and masks

void make_pieces(){
    create_piece( 0, 1, 2, 3, 4, 0, 0, 0, 0, 0 );
    create_piece( 0, 1, 2, 3, 0, 0, 0, 0, 0, 1 );
    create_piece( 0, 1, 2, 3, 1, 0, 0, 0, 0, 1 );
    ...
}

Procedure create_piece

- initializes the piece
- tries to make masks for the piece according to the original orientation, its mirror image, and both rotated in 90 degrees steps

A mask is suggested for use with the procedure suggest_mask( mask_type &mk )

- normalizes the coordinates of the mask
- rejects the mask, if the piece already has an equivalent mask
- otherwise adds the mask to the mask table, sets width, height, and depth, and increases the number of masks of the piece
The coordinates of the mask are normalized in order to

- recognize equivalent masks as the same
  - e.g., [ (0,0), (1,0), (2,0), (3,0), (4,0) ] and [ (0,0), (4,0), (2,0), (3,0), (1,0) ]
- ensure that square (0,0) is reserved
- ensure that the $x$-coordinates are positive or zero
- avoid squares of the form (0, negative)

Different representations of the same object can be investigated with the notions of

*isomorphism* and *equivalence class*

- the information is thrown away that is related to the representation of, but not relevant for, the actual object

- another example: when are two triangles \([(x_1,y_1),(x_2,y_2),(x_3,y_3)\)] the same? How about two polygons with 5 corners?
The normalization steps

- sort the squares primarily according to their \(x\)-coordinates and secondarily according to their \(y\)-coordinates

- subtract \(m_x[0]\) from the \(x\)-coordinate of each square (except if \(m_x[0] = 0\)), and similarly with the \(y\)-coordinates

Establishes

\[
\begin{align*}
m_x[0] &= m_y[0] = 0 \\
\forall i: & 0 \leq m_x[i] \\
\forall i: & m_x[i] = 0 \rightarrow m_y[i] \geq 0 \\
\forall i: & m_x[i] \leq m_x[i+1] \\
\forall i: & m_x[i] = m_x[i+1] \rightarrow m_y[i] \leq m_y[i+1] \quad \text{(or } m_y[i] < m_y[i+1])
\end{align*}
\]

\(m_x[i]\) has its maximum value when \(i = \text{size_of_piece} - 1\)

Now a mask can be recognized as already existing by comparing it square by square to the other masks of the same piece

- at most 7 masks to compare with

\[\Rightarrow\] there is no point in using, e.g., a hash table or a map container
3.6 The backtracking algorithm

A problem: when backtracking, the mask that should be removed is not necessarily the one that is met first.

A potential solution: mark the square 0 of the mask differently from other squares on the board.

Another solution: keep track of the used masks with a stack that is implemented as a linked list:

- simple and efficient
- the need for extra memory is negligible
- facilitates jumping directly to the right square when backtracking, if the coordinates of the square have been stored in the mask or piece.

After wiping out the mask from the board, it is in any case necessary to visit the piece, to release it.

⇒ Implementation: the variable `previous` and the field `previous` in piece records comprise a linked list that goes mask → piece → mask → piece …
void trial_and_error(){
    int bx = 0, by = 0, // the square that the program tries to fill
         previous = -1, // the number of the most recently used mask
         next_try = 0; // the mask that is tried next
    int nr_solutions = 0;

    while( true ){
        /* Try to make progress */
        do{
            if( the_board[ bx ][ by ] ){
                /* Pass by the reserved square */
                ++by;
                if( by >= y_size ){
                    by = 0; ++bx;
                    if( bx >= x_size ){
                        ++nr_solutions; process_a_solution();
                        next_try = masks;
                    }
                }
            }
        }while( the_board[ bx ][ by ] );
    }
}
} else {

    /* Try to put a new piece */
    while (
        next_try < masks && !put_mask( next_try, bx, by )
    ) {
        ++next_try;
    }
    if ( next_try < masks ) {
        piece[ mask[ next_try ].piece_nr ].previous = previous;
        previous = next_try;
        next_try = 0;
    }
}

} while ( next_try < masks );
/ Backtrack */

if( previous != -1 ){
    piece_type &pc = piece[ mask[ previous ].piece_nr ];
    bx = pc.nx; by = pc.ny;
    next_try = previous; previous = pc.previous;
    remove_mask( next_try ); ++next_try;
}else{ break; }

}
3.7 Observations

The information that is stored was chosen and placed onto the board, piece, or mask so that testing whether a mask could be put on the current square, putting it there, and removing it would be as fast as possible

- where it was helpful, "extra" information was stored: width, height, depth

- this was worthwhile, because masks are tested, put and removed very many times

The nature of the information helped a bit in choosing its place

- reserved or not is a property of a piece and not of a mask

A fast access path was designed for the information

- the number of the previous mask is found from the piece that is being removed

- the mask stores the number of the corresponding piece

There was only a small amount of changing information, and none in the masks
The representation of masks was normalized so that

- it would be easy to test of a new candidate mask whether it is a copy of an already existing one

- the mask would extend above, above right, right, and down right from the reference square, and not to other directions

—— End of case 2 ——