Backtracking

Prof. Rhadamés Carmona

Last revision: 28/10/2019

Concept

- Introduced by Lehmer, 50's. R.J. Walker described it algoritmically, 1960. Later, it was developed by S. Golamb y L. Baumert.
- Sistematically and Incrementally Backtracking explores the solution space, considering on each step all the options. It rejects candidates when it is not posible to complete a solution. Reduce the amount of work in large searches.

Concept

- The algorithm extends a partial solution considering all possible solution paths. These partial solutions form a potential search tree, which is traversed top-down in "depth-first" order (DFS or in depth).
- Those branches where it is known that no solution is reached, are pruned, and so the tree does not grow indefinitely.

Examples

- Knapsack: maximize value of carried items with the limitation of a maximum weight
- Sudoku: place digits without repetition (row, column, 3x3 submatrix)
- 8 queens: put 8 queen in a chess board in safe positions. Can be generalized for n queens.
- Rat in maze: starting at the cell (1,1), find the output (n,n) in a labyrinth with obstacles
- Given an array of n elements, generate permutations P(n), variations V(n,m) and combinations C(n,m)
- M-coloring: color a graph with m colors, where adjacent nodes need to have different colors.

Template algorithm

```
void bt(partial_solution, candidates)
{
    if (!is_valid( partial_solution)) return;
    if (is_solution(partial_solution))
        process_solution(partial_solution);
    /* else ?*/
    for each x in candidates
        bt(partial_solution⊕x, candidates – {x} );
}
```

Template algorithm

- **is_valid(s)**: partial solution **s** should satisfy the problem constraints. Avoid unnecessary searching when it is not posible to find a solution in further steps from **s**. Example, when we have 7 queens, and all of them are in safe position, is_valid is true (even though it is not a solution with 8 queens). But, if a queen is not safe, we cannot build a solution from **s** in further steps, and is_valid = false.

Template algorithm

- **is_solution(s)**: It is true when it can be considered as a possible solution. Example, when the last element of **s** coincides with the output of the maze (rat in a maze), or when |s|=8 and all the queens are safe (8-queens problem)

Template algorithm

- **bt(partial_solution⊕x, candidates – {x})**: we add x to the partial solution, while x is not a candidate in deeper recursive levels.
- In some specific problems, x can be added several times. Then, we cannot exclude it from candidates.

Template algorithm

- **bt(partial_solution⊕x, candidates – {x})** can be written like this

```
If (is_valid(partial_solution + x)) // avoid a recursive call
{
  partial_solution += x;          // add alternative
  candidates -= {x};              // exclude alternative (depends)
  bt(partial_solution, candidates);
  candidates += {x};              // include alternative (depends)
  partial_solution -= x;          // remove alternative
}
```

Template algorithm

- **process_solution(c):** Depending on the problem, it is possible to exit after finding the first solution (use a global logic "found" to abort the search at all levels). But if you want the optimal solution, then you can compare this solution with the best one found. And if you want all the solutions, you can save or print the solution at this time.

Knapsack

- Given N objects, with a value and a weight (each one), fill the Knapsack without exceeding its capacity (MAX weight) but maximizing the sum of their values
- **is_valid**: the sum of the weights should not exceed MAX
- **is_solution**: any valid no-empty knapsack is a posible solution

Knapsack

- **Partial_solution**: a list of objects with the sum of their weights, and the sum of their values.
- **Candidates**: all the remaining objects that may be included in the knapsack.

```
struct element {
    public:
        int value, weight;
}
```

Knapsack

```
class solution : public list<element> {
    public:
        int svalue=0, sweight=0;
};

solution knapsack(vector<element> &candidates, int max) {
    solution current, best;
     knapsack(0, current, candidates, best);
    return best;
}
```

Knapsack

```
void  knapsack (int step, solution &s, vector<element>
    &candidates, solution &best) {
    if (c.sweight > MAX) return;
    if (s.svalue > best.svalue) best = s;
    for (int i=step; i<candidates.size(); i++) {
      const element &x = candidates[i];
      s.push_back(x); s.sweight +=x.weight; s.svalue+= x.value;
      knapsack (step+1, s, candidates, best);
      s.pop_back(); s.sweight -=x.weight; s.svalue-= x.value;
    }
}
```

Permutations

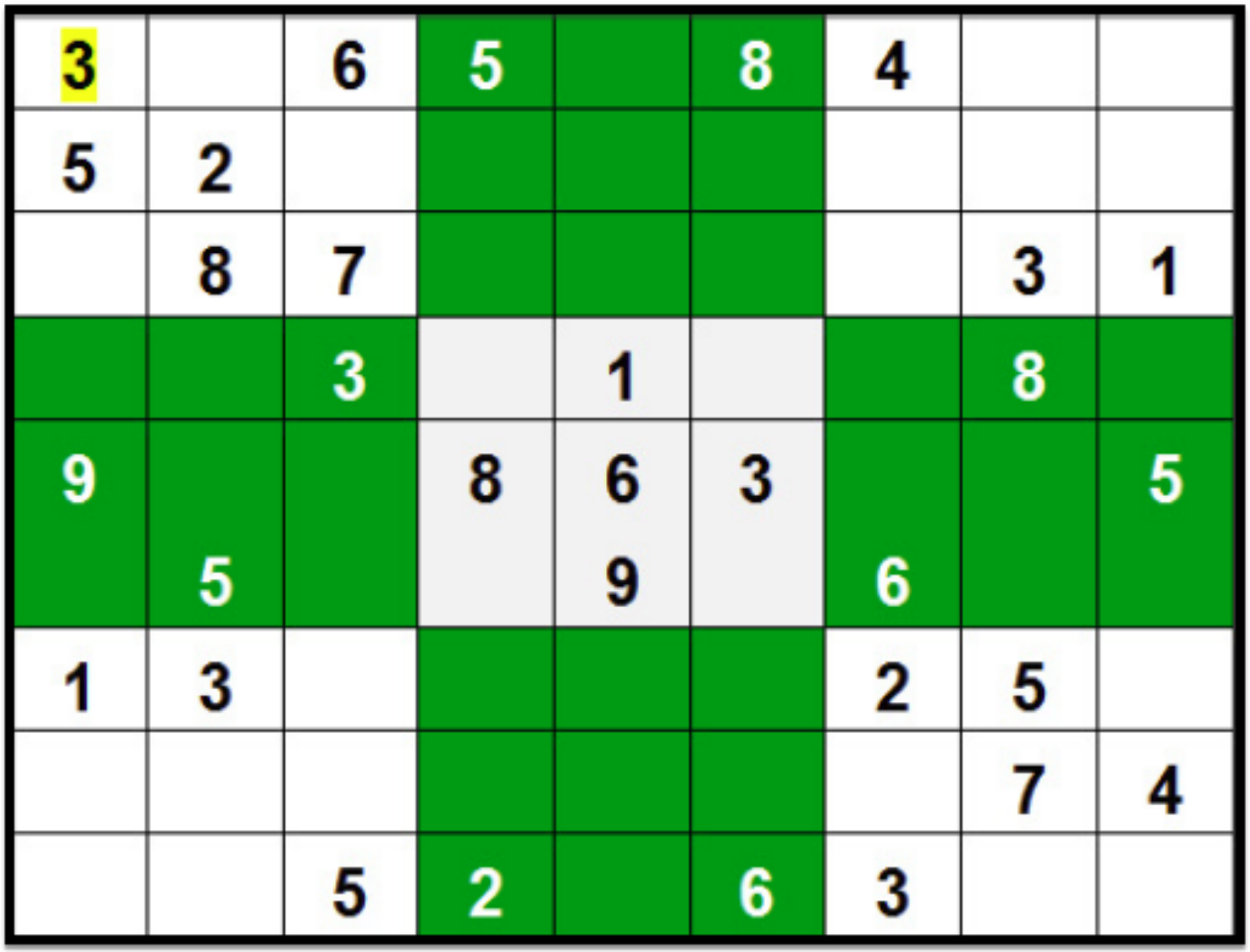
```
void bt(sol, candidates)
{
  if (!is_valid( sol )) return;
  if (is_solution( sol ))
    process_solution( sol );
  /* else ?*/
  for each x in candidates
      bt(partial_solution⊕x, candidates – {x} );
}
```

0 1 2 3 1 0 2 3 2 1 0 3 3 1 2 0
0 1 3 2 1 0 3 2 2 1 3 0 3 1 0 2
0 2 1 3 1 2 0 3 2 0 1 3 3 2 1 0
0 2 3 1 1 2 0 3 2 0 3 1 3 2 0 1
0 3 2 1 1 3 2 0 2 3 0 1 3 0 2 1
0 3 1 2 1 3 0 2 2 3 1 0 3 0 1 2

```
// We don't really need P, we can use A
void permute(int step, int A[], int P[], int n)
{

  if (step==n)
     print(P,n);
  else
    for (int i=step; i<n; i++)
    {
              P[step] = A[i];
              std:: swap(A[step], A[i]);
              permute(step+1, A, P, n);
              std::swap(A[step], A[i]);
    }
}
```

Sudoku



Every colum, row and 3x3 square should contain all digits 1..9

Sudoku

```
void sudoku(int A[9][9], list<pair<int,int>> &unprocessed, bool &found){
  if (unprocessed.size()==0)
  {
     print(A); found = true;
  }
  else
  {
    int i = unprocessed.begin()->first;
    int j = unprocessed.begin()->second;
    unprocessed.pop_front();
    for (int digit=1; digit<=9 && !found; digit++)
   {
                A[i][j] = digit;
                if (horizOK(A, i) && vertOK(A,j) && subMatrixOK(A,i,j))
                            sudoku(A, unprocessed, found);
                A[i][j] = 0;
    }
  }
}
                                  //Complexity = 9^n, n = #free positions.
```

Rat in maze

```
void rat (int i, int j, int A[n][n], list<pair<int,int>> &path) {
   if (i==n-1 && j==n-1)
          process_solution(A, path);
   else   {
             pair<int,int> candidates[4] = { {i-1,j}, {i+1,j}, {i,j-1}, {i,j+1} }; // pseudocode
             for (each x in candidates)   {
                        int a = x.first;
                        int b = x.second;
                        if (a>=0 && a<n && b>=0 && b<n && A[a][b]==false) {
                                   A[a][b] = true;
                                   path.push_back(make_pair(a,b));
                                   rat(a,b,A,path);
                                   path.pop_back();
                                   A[a][b] = false;
                        }
             }
    }
}
```

Advantages

- Warranty: it finds the solution, if it exists.
- It is simple to implement, short code, and the solutions usually follow the same scheme.
- Incrementally explore all possible solutions, and early discard candidates that will not lead to the solution.
- Get all possible solutions.

Drawback

- There is no memorization, so you can go through a potential search subtree many times.
- It is usually of an “exponential” nature, which makes it impractical for large problems.
- We need smart pruning to reduce its execution time.
- It consumes a lot of stack space, in medium or large problems.
- The more alternatives per state, the slower.

Spoj problems

- 224 (Vonny and her dominos)
- 4525 (Digger Octaves)
- 1538 (Making Jumps)
- 15609
- 28587

Spoj problems

224 (Vonny and her dominos)

- Read the board
- For each stone (28 stones), pre-compute where it can be located in the board (visit the board in pairs, and update the vector of positions per stone)
- Do the backtracking per stone. On each recursive level, consider all posible positions of only one stone.
- Recursive base case: when you have put 28 stones (increment a global counter)

Spoj problem

4525 (Digger Octaves)

- Call backtracking from each (i,j), with a(i,j)='X'
- Generate all paths, starting from (i,j)
- When a path has a length of 8, check if it was previously registered; here the magic!
- We need a fast way to compare paths; I store each path in one "set<coordinate>", where the coordinates are lexicographically sorted (by x, and them by y-axis). A compare function has been developed to store unique paths (and no duplicated paths) in a set of paths

Spoj problem

1538 (Making Jumps)

- I have tried to solve it (unsuccessfully…grrrr….)
- I think the problem statement is a bit confusing
- There are small tricks or details in the output, like (squares/square, can not)
- Take care with this: “with each row offset zero or more columns to the right of the row above it”
- Also, with this “not reach in any number of moves without resting in any square more than once?”. Unreachable nodes are counted or not?

Spoj problem

Good luck!