Lecture Notes on
Data Structures and Algorithms: Exhaustive Generation & Search

Conrado Martínez
U. Politècnica Catalunya

February 17, 2016

Part VI

Exhaustive Generation & Search

Introduction

In many computational problems we must exhaustively explore a solution space. Many other times, it is the only way we know to solve the problem at hand.
Some examples:

- Generating all combinatorial objects of a given size, such as bitstrings, permutations, trees, graphs, . . .
- Finding a non-menacing configuration of n queens in a $n \times n$ chessboard
- Finding a shortest Hamiltonian cycle in a weighted graph (Traveling Salesman Problem)
- and many more . . .

Introduction

In more general terms, we look at computational problems in which solutions are n-tuples $x = (x_1, \dots, x_n) \in D_1 \times \cdots \times D_n$ which satisfy some additional constraints. The potential solution space is $D_1 \times \cdots \times D_n$ (typically of exponential size in n or larger), but it might contain no or very few solutions. In general we will be interested in

- Finding all solutions
- Finding if at least one solution exists (and return such a solution)
- Finding an optimal solution according to some criterion (maximizing a benefit/minimizing a cost)

Introduction

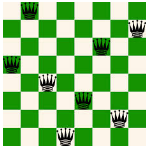
Our task is thus to take a sequence of decisions, setting the value of x_1, ..., x_n until a solution is found. Depending on the task we must perform we might need to continue (finding all solutions, finding a better solution, if it exists).

Brute force generates n-tuples $x = (x_1, \ldots, x_n) \in D_1 \times \cdots \times D_n$ and checks for each tuple whether it is a solution or not. The process stops when we have generated all tuples or when a solution has been found.

Part VI

Exhaustive Generation & Search

Example: n Queens



Given an $n \times n$ chessboard, is there a way to lay n queens in the board in such a way that no queen menaces any other? Find one such configuration if it exists.

Example: n Queens

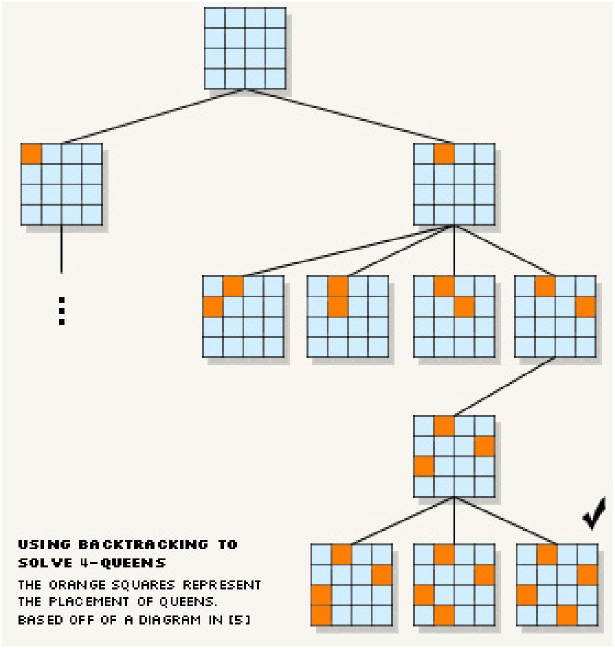
- $D_i = \{1, \ldots, n^2\}$. Check n^{2n} configurations. $64^8 =$ 281 474 976 710 656.
- Every queen must be in a different row, $D_i = \{1, \ldots, n\}$. Check n^n configurations. $8^8 =$ 16 777 216. Assuming that the n-tuples are generated in lexicographic order, first solution found after 1 299 852 steps.
- Every queen must be in a different row and column, $|D_i| = i$ if we assign queens from row n to row 1. Check $n!$ configurations. $8! =$ 40 320, first solution found after 2 830 steps.
- This chapter: we need only to examine at most 2 057 configurations. First solution found in 114 steps.

Example: n Queens

Using backtracking, we explore (in a DFS-like manner) the space solution, building partial solutions and

1. extending the current partial solution by making an additional decision if the current partial solution does not violate the constraints (it is feasible)
2. undoing the last decision (backtracking) if the current partial solution does not satisfy some constraint (feasibility pruning)

Example: n Queens


USING BACKTRACKING TO SOLVE 4-QUEENS
THE ORANGE SQUARES REPRESENT THE PLACEMENT OF QUEENS.
BASED OFF OF A DIAGRAM IN [5]

Example: n Queens

```
// sol[i] = column of the queen in the i-th row, 1 <= k <= n
// Pre: 0 <= k <= n+1, sol[1..k] is a feasible partial solution
// Post: returns true iff a non-menancing configuration exists, sol represents
//       such a configuration
bool queens(vector<int>& sol, int k, int n) {
     if (k == n)
        return true;
     // k < n
     bool sol_found = false;
     for (int j = 1; j <= n and not sol_found; ++j) {
         sol[k+1] = j;
         if (feasible(sol,k+1))
            sol_found = queens(sol, k+1, n);
        // else prune
     }
     return sol_found;
}
```

Example: n Queens

```
// we need only to check if the last queen at row p
// menaces some of the previous queens (they do not menace between them)

bool feasible(const vector<int>& sol, int p) {
  for (int q = 1; q < p; ++q) {
      if (sol[q] == sol[p]) return false;   // same column
      if (sol[q] - q == sol[p] - p) return false; // same SW -> NE diagonal
      if (sol[q] + q == sol[p] + p) return false; // same NW -> SE diagonal
  }
  return true;
}
```

Example: n Queens

A common trick is to encapsulate the backtracking algorithm and auxiliary data structures in a class:

```
class NQueens {
public:
      NQueens(int n = 8);
      vector<int> solve() const; // returns (-1,-1,...,-1)
                                 // if no solution exists
private:
      int n_;
      vector<int> sol_; // sol_[0] not used
bool backtrack(int k);
bool feasible(int p) const;
}

NQueens::NQueens(int n) : n_(n), sol_(n+1, -1) {
         backtrack(0);
}
1
vector<int> NQueens::solve() const {
        return sol_;
}

bool NQueens::backtrack(int k) {
     if (k == n_)
        return true;
     bool sol_found = false;
     for (int j = 1; j <= n_ and not sol_found; ++j) {
        sol_[k+1] = j;
        if (feasible(k+1)) ...
     }
     return sol_found;
}
```

Example: Subset Sum

Given a set $S = \{S_0, \ldots, S_{n-1}\}$, of n integers, is there a partition of S such that the sum of the elements on each part is the same, that is, is there a subset $A \subseteq S$ such that

$$\sum_{x \in A} x = \sum_{x \notin A} x?$$

Example: Subset Sum

We will store the elements of S in a vector, and represent solutions (partitions of S) by means of its characteristic function (a Boolean vector):

```
vector<int> S(n);
vector<bool> inA(S.size());
```

with $inA[i] =$ **true** iff $S[i] \in A$.

In level k of the backtracking we try both possibilities: $S[k] \in A$ and $S[k] \notin A$.

Example: Subset Sum

```
class SubsetSum{
public:
      SubsetSum(const vector<int>& S);
// returns true iff the problem has a solution; in that case, it
// also returns the characteristic function of $A$ and the sum
      bool has_solution(int& sum, vector<bool>& inA) const;
private:
      int n_;
      vector<int> S_;
      vector<bool> inA_;
      bool has_sol_;
      int sum_inA_;              // <=== marks
      int sum_not_inA_;
      int sum_;

bool backtrack(int k);
}

SubsetSum::SubsetSum(const vector<int>& S) : n_(S.size()), S_(S),
         inA_(n_), has_sol_(false), sum_inA_(0), sum_not_inA_(0)
{
         for (int i = 0; i < S.size(); ++i) sum_ += S[i];
         has_sol_ = backtrack(-1);
}
```

Example: Subset Sum

```
bool SubsetSum::backtrack(int k) {
     if (k == n_)
        return sum_inA_ == sum_not_inA_;
     sum_ -= S[k+1];        // <= marking
     // try S[k+1] in A
     inA_[k+1] = true;
     sum_inA_ += S[k+1];     // <= marking
     if (sum_inA_ <= sum_ + sum_not_inA_)
        has_sol_ = backtrack(k+1);
     // if no solution found, try S[k+1] not in A
     if (not has_sol_) {
       inA_[k+1] = false;
       sum_inA_ -= S[k+1]; // <= unmarking
       sum_not_inA_ += S[k+1]; // <= marking
       if (sum_not_inA_ <= sum_ + sum_inA_)
          has_sol_ = backtrack(k+1);
    }
    if (not has_sol_) {
       sum_notinA_ -= S[k+1]; // <= unmarking
       sum_ += S[k+1];        // <= unmarking
    }
}
```

Marking

Marking/unmarking is the use of auxiliary memory (`sum_`, `sum_inA_`, etc. in the previous example) to avoid computations. Information passes from the top levels to the bottom levels of the recursion (marking). When we backtrack it is necessary to update/remove the marks (unmarking).

When marks are passed through value parameters the unmarking isn't necessary; but if marks are passed by-reference or are globally accessible attributes then we need to take care of the unmarking.

Example: Knapsack

Given a set S of n objects with weights $w_0, \ldots, w_{n-1}$ and values $v_0, \ldots, v_{n-1}$, and a knapsack capacity $C > 0$, we want to find a subset S' of objects with total weight $\leq C$ and maximizing the total value; that is, find $S' \subseteq \{0, \ldots, n-1\}$ such that

$$\sum_{i \in S'} w_i \leq C$$

$$\sum_{i \in S'} v_i \text{ is maximum}$$

Example: Knapsack

```
struct Object {
   double weight, value;
   string name;
}

class Knapsack {
private:
   vector<Object> Obj;    // input
   double C;              // input
   int n;                 //
   vector<bool> sol;      // current partial solution
   vector<bool> optsol;   // best solution found
   double val, optval;    // current value, best value
   double w;              // current weight

double estimation(int k);
void backtrack(int k);
void find_approximate_optimal_solution();

public:
   ...
};
```

Example: Knapsack

```
bool  by_decr_value_weight_ratio(const Object& a, const Object& b) {
    return a.value/a.weight > b.value/b.weight;
}

class Knapsack {
private:
  ...
public:
  Knapsack(const vector<Object>& objects, double capacity) :
    Obj(objects), C(capacity), n(Obj.size()), sol(n), optsol(n,false),
    val(0.0), optval(0.0), w(0.0) {

    sort(Obj.begin(), Obj.end(), by_decr_value_weight_ratio);
    find_approximate_optimal_solution();
    backtrack(0);
  };
  void get_optimal_solution(vector<Object>& optimal_sol, double& optimal_value) {
    optimal_sol = vector<Object>();
    for (int k = 0; k < optsol.size(); ++k)
        if (optsol[k]) optimal_sol.push_back(Obj[k]);
    optimal_value = optval;
  }
};
```

Example: Knapsack

```
void Knapsack::backtrack(int k) {
    if (k == n) {
       if (val > optval) {
          optval = val;
          optsol = sol;
       }
    } else {
      sol[k] = true;
      val += Obj[k].value;
      w += Obj[k].weight;
      if (w <= C and Âestimation(k) > optval})
         backtrack(k+1);
      sol[k] = false;
      val -= Obj[k].value;
      w -= Obj[k].weight;
      if (w <= C and Âestimation(k) > optval})
         backtrack(k+1);
    }
}
```

Example: Knapsack

`estimation` is a heuristic that gives us an upper bound on the total value that we could obtain with the current decisions taken so far (as expressed in `sol`).

The idea is simple: the total value can be at most val (the current accumulated value), plus the value of all remaining objects

```
double Knapsack::estimation(int k) {
int est = val;
for (int i = k+1; i < n; ++i)
    est += Obj[i].value;
}
```

If at some point of the backtracking the estimation for a current partial solution is not larger than $optval$ (the best value found so far) then it is worthless exploring that part of the solution space (current best solution pruning, CBSP).

Example: Knapsack

Ordering objects by decreasing ratio value/weight is helpful to find better solutions at an early stage and take advantage of CBSP as soon as possible. An approximate optimal solution can be easily found by putting objects in the knapsack by decreasing ratio value/weight, until not enough capacity remains.

```
void Knapsack::find_approximate_optimal_solution() {
   double currw = 0.0; int k = 0;
   while (currw + Obj[k].weight < C) {
      optsol[k] = true;
      optval += Obj[k].value;
      currw += Obj[k].weight;
      ++k;
   }
}
```

Current Best Solution Pruning

Current best solution pruning (CBSP) is a useful technique that can be applied whenever we want to solve optimization problems (maximization or minimization).

Assume we are trying to find a solution of minimum cost; let x be the current solution, and `best_cost` the cost of the best solution found so far. What we need is an estimation of the cost of the best solution that we can find from x; it must be a lower bound of the cost of the best solution that we can find from x:

$$\texttt{estimated\_cost}(x) \leq \texttt{real\_best\_cost}(x)$$

Current Best Solution Pruning

Then if

$$\texttt{estimated\_cost}(x) > \texttt{best\_cost}$$

we can prune at x and avoid exploring its descendants in the recursion tree, because no solution found there will be better than the one we already have.

For maximization problems, the estimated benefit must be an upper bound to the real benefit

$$\texttt{estimated\_benefit}(x) \geq \texttt{real\_best\_benefit}(x)$$

and CBSP can be applied if

$$\texttt{estimated\_benefit}(x) < \texttt{best\_benefit}$$

Current Best Solution Pruning

In most situations the estimation is the sum of two contributions: the cost/benefit of the current partial solution, plus the cost/benefit $h(x)$ coming from the remaining decisions.

Marking is useful to pass information about the current cost/benefit from one level to other, and we need only to compute $h(x)$ (this can also be avoided sometimes using marking). The function $h(x)$, called the heuristic, must be

1. Faithful: for minimization problems we must have

   $$h(x) \leq \texttt{real\_cost}(x) - \texttt{cost}(x)$$

   for any partial solution x.
2. Easy to compute.

Example: Graph Coloring

Given an undirected graph $G = \langle V, E \rangle$, find its chromatic number $\xi(G)$, the minimu number of colors in a valid coloring of G.

A valid coloring of G with k colors is a function $c : V(G) \to \{0, \ldots, k-1\}$ such that $c(u) \neq c(v)$ whenever $(u, v) \in E$.

Example: Graph Coloring

```
// We assume V(G) = {0,...,n-1}
class Coloring {
public:
    Coloring(const Graph& G);
int chromatic_number() const;
    ...
private:
    Graph G_;
    vector<int> color;
    vector<int> best_coloring;
    set<int> colors;
    int chromatic_nr;
void backtrack(vertex k);
    ...
};

Coloring::Coloring(const Graph& G) : G_(G),
          color(G_.nr_vertices(),-1),
          best_coloring(G_.nr_vertices()),
          chromatic_nr(G_.nr_vertices()) {
      for (int i = 0; i < G_.nr_vertices(); ++i)
           best_coloring[i] = i;
      color[0] = 0;
      colors.insert(0);
      backtrack(1);
}
```

Example: Graph Coloring

```
void Coloring::backtrack(vertex k) {
  if (k == G_.nr_vertices()) {
     if (colors.size() < chromatic_nr) {
       chromatic_nr = colors.size();
       best_coloring = color;
     }
  } else {
     vector<bool> used(colors.size(),false);
     forall_adj(w, G_[k]) { // for all w adjacent to k in G_
        if (color[w] != -1) used[color[w]] = true;
     }
     // try coloring vertex k with some color not used by
     // any neighbor
     for (int i = 0; i < used.size(); ++i) {
       if (not used[i]) {
           color[k] = i;
           backtrack(k+1);
           color[k] = -1;
       }
     }
     // try coloring vertex k with a new color
     int i = colors.size();
     color[k] = i;
     colors.insert(i);
     backtrack(k+1);
     color[k] = -1;
     colors.remove(i);
  }
}
```

Example: The Traveling Salesman Problem

Given a weighted (un)directed graph $G = \langle V, E \rangle$ the Traveling Salesman Problem (TSP) asks to find the minimum weight Hamiltonian cycle in G.

This problema has a vast range of applications and it is the archetype of computationally hard combinatorial optimization problem.

People have developed a vast array of efficient methods to get good approximate solutions, and have also considerably reduced the time to exactly solve many practical instances, with thousands of vertices, despite exact solutions require exponential time in the worst case.

Example: TSP

```
struct arc {
       vertex src, tgt;
       double weight;
};
typedef vector< list<arc> > Graph;
typedef int vertex;

class TSP {
private:
   Graph G_;
   int n;
   vector<arc> arc; // current partial solution =
                    // a sequence of arcs
   double curr_weight; // current weight
   vector<bool> visited; // vertices visited by current
                         // partial solution
   vector<arc> best_tour; // best solution
   double best_weight;
   ...
public:
   TSP(const Graph G) : G_(G), n(G_.nr_vertices()),
                        arc(n), curr_weight(0.0),
                        visted(n, false) {
      // try to find a good approximate solution using
      // some simple heuristic, e.g., DFS trying first shortest edges
      // to initialize best_tour and best_weight

      // start with fictitious loop at vertex 0
      backtrack(0, arc(0,0, 0.0));
   }
}
```

Example: TSP

```
void TSP::backtrack(int k, arc e) {
  vertex v = target(e);
  if (k == n) {
     if (v = source(arc[0]))
        if (curr_weight < best_weight) {
           best_tour = arc;
           best_weight = curr_weight;
        }
  } else {
    arc[k] = e;
    visited[v] = true;
    curr_weight += weight(e);
    forall_succesors(e', G_[v]) {
       vertex w = target(e');
       if ((not visited[w] or k == n-1) and
           curr_weight + weight(e') + estimated_weight(w) < best_weight)
           backtrack(k + 1, ep);
    }
    curr_weight -= weight(e);
    visted[v] = false;
  }
}
```

Example: TSP

`estimated_weight(w)` must give us a lower bound on the weight of any simple path connecting vertex w with vertex 0, such that all non-visited vertices belong to the path and no visited vertex (except 0) belongs to the path.

The trivial function `estimated_weight(w)` $= 0$ does the job, but we can do much better with a little more effort.

Example: TSP

To find the desired lower bound, one idea is to relax the requirement that we find a simple path. A minimum “spanning” tree with vertex set $V' = \texttt{non-visited} \cup \{0\}$ can be computed using an easy variation of Prim’s algorithm starting from vertex w. Such a tree touches all non-visted vertices (including w) and no visted vertex, its weight is a lower bound for the weight of any simple path connecting w with 0 of mimimum weight, passing through all non-visited vertices, and not using any visited vertex (except 0).

Example: TSP

Another easy way to find a lower bound is to find for every non-visited vertex the arcs with minimum weight that enter and leave such vertex, add the two weights, and divide by 2. For w we shall do the same but just with the arc of minimum weight leaving from w (and divide by 2), and for vertex 0 (the origin of the tour by convention) we add the weight of the arc of minimum weight that enters 0, divided by 2.

Backtracking: Recap

```
▷ $x$ is the current partial solution
procedure BACKTRACK($k$)
    if IS_SOLUTION($x$) then
        PROCESS($x$)
    else
        for $v \in D_k$ do
▷ exit from the loop if we are looking
▷ for one solution and it has been found
            $x[k] := v$
            MARK($x, v$)
            if IS_FEASIBLE($x, k$) then
                BACKTRACK($k + 1$)
            else ▷ feasibility pruning + CBSP if optimizing
            end if
            UNMARK($x, v$)
        end for
    end if
end procedure
```

Part VI

Exhaustive Generation & Search

Branch & Bound

Backtracking is a blind search scheme, since the order of exploration of the configuration trees is fixed beforehand.

Branch & Bound is an informed search scheme since it explores the configuration tree in order of estimated costs (or estimated benefits). The most promising areas of the configuration tree are explored before other areas. Since B&B is usually finding better solutions before Backtracking does, B&B exploits the “Current Best Solution Pruning” much more than Backtracking does.

Branch & Bound

B&B is more costly in memory usage. In BT a current partial solution x encodes a path from the root to the currently explored node, and behaves as a stack.

In B&B each node must be explicitly represented, and it must contain all the information of the current partial solution it represents, and perhaps

- its level
- the cost or benefit of the current partial solution
- other marks

Branch & Bound

B&B must also maintain a list of alive nodes. At each iteration of the B&B, an alive node is selected, namely, the one with smallest estimated cost (the most promising).

Then the node is expanded: all its feasible successors are generated and added to the list of alive nodes. The current node then dies (and gets removed from the list of alive nodes).

Since many nodes might be alive at a given stage of the B&B, the memory consumption is high, for all alive nodes must be kept and each node is a relatively large object.

Branch & Bound

The information explicitly kepy at each node is used to generate all its successors. For example if y is a node it can contain a vector $x = (x_0, \ldots, x_{k-1})$ with the decisions made for the first k variables; each successor y' will contain a vector $x' = (x_0, \ldots, x_{k-1}, x'_k)$ with just one more decision, namely, the $(k + 1)$-th variable takes value x'_k.

When a new best solution is found, the list of alive nodes should be purged: alive nodes with estimated cost larger than the new best cost can be removed. It is not a must (these nodes will be removed and not expanded thanks to CBSP), but it is useful to purge the list whenever possible to keep the list as small as possible.

Branch & Bound

```
procedure BRANCH-AND-BOUND($z$)
    $Alive$: a priority queue of nodes
    $y :=$ ROOT_NODE($z$)
    $Alive$.INSERT($y$, ESTIMATED-COST($y$))
    Initialize $best\_solution$ and $best\_cost$, e.g., $best\_cost := +\infty$
    while $\neg Alive$.IS_EMPTY() do
        $y := Alive$.MIN_ELEM(); $Alive$.REMOVE_MIN()
        for $y' \in$ SUCCESSORS($y, z$) do
            if IS_SOLUTION($y', z$) then
                ...
            else
                $feasible :=$ IS_FEASIBLE($y', z$)$\wedge$
                        ESTIMATED-COST($y'$) $< best\_cost$
                if $feasible$ then
                    $Alive$.INSERT($y'$, ESTIMATED-COST($y'$))
                end if
            end if
        end for
    end while
end procedure
```

Branch & Bound

```
...
if Is_Solution($y', z$) then
    if Cost($y'$) $< best\_cost$ then
        $best\_cost :=$ Cost($y'$)
        $best\_solution := y'$
        Purge nodes with larger priority
    end if
else ...
end if
...
```