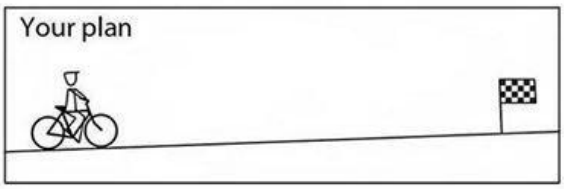
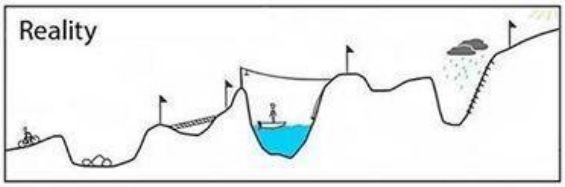
Section 5 – Backtracking

- *5.1 Introduction.*
- *5.2 General Schema.*
- *5.3 Analysis of execution time.*
- *5.4 Examples:*
  - *5.4.1 Exhaustive search in graphs.*
  - *5.4.2 The 0/1 Rucksack problem.*
  - *5.4.3 The Salesman problem.*
  - *5.4.4 Task planning.*
  - *5.4.5 Other examples.*

1 – Introduction





Backtracking algorithms
The algorithm correct solution on the way

1 – Introduction

Characterization of the problems to be solved with backtracking:

The problems whose resolution we are going to address, have characteristics that allow us to work as follows:

- We look for a solution, the best solution or the set of all solutions that satisfy certain conditions.
- Each solution is the result of a sequence of decisions.
- There is an objective function that must be satisfied by each selection, or optimized if we only want the best.

- If it is known an optimal selection criterion for each decision step:

Greedy approach

1 – Introduction

Characterization of the problems to be solved with backtracking:

The problems whose resolution we are going to address, have characteristics that allow us to work as follows:

- We look for a solution, the best solution or the set of all solutions that satisfy certain conditions.
- Each solution is the result of a sequence of decisions.
- There is an objective function that must be satisfied by each selection, or optimized if we only want the best.

- If Bellman's optimality principle is fulfilled and the partial solutions can be structured in an acceptable table:

Dynamic Programming approach

E.D.A. II – Fernando Bienvenido – Curso 2020/21

1 – Introduction

Characterization of the problems to be solved with backtracking:

The problems whose resolution we are going to address, have characteristics that allow us to work as follows:

- ➢We look for a solution, the best solution or the set of all solutions that satisfy certain conditions.
- ➢Each solution is the result of a sequence of decisions.
- ➢There is an objective function that must be satisfied by each selection, or optimized if we only want the best.

- If it is not possible to solve this type of problem using the previous techniques (greedy or dynamic programming) :

IT IS REQUIRED TO SEACH SOLUTIONS *Backtracking* approach

1 – Introduction

Characterization of the problems to be solved with backtracking:

If the solution to our problem involves making a series of decisions, but:

- ✓ We do not have enough information to know which one to choose a priori.
- ✓ Each decision leads us to a new set of decisions.
- ✓ One sequence of decisions (and maybe more than one) can solve our problem.
- ✓There may be decision sequences (often many) that are not a solution (leading us to dead ends).

In this situation, it would be desirable to explore a sequence of decisions, and if we see that it does not lead to the solution, go back and change it.

1 – Introduction

Characterization of the backtracking approach:

Backtracking algorithms (*back* or *return* algorithms).

- Search for the/a solution, through a systematic exploration of all possible solutions to the problem.
- A sequence of decisions is explored, and if it does not lead us to the solution, we go back and change it.
- These algorithms do not follow any rules to find the solution, they just try everything possible, until they find the solution or find that there is no solution to the problem.
- The search is composed by several partial searches or subtasks (sequence of decisions).
- These subtasks often include more subtasks, so the general treatment of these algorithms is naturally recursive.

1 – Introduction

Formal characterization of problems solvable with backtracking:

- The solution is built in steps, that is, it can be expressed in the form of an n-tuple, (x1, x2, x3, ..., xn), where each xi belongs to a finite set Si, and represents the decision made in the i-th step, from a finite set of alternatives.

- The problem can be formulated as the search of the n-tuple that optimizes (maximizes or minimizes), or simply satisfies, a certain criterion (predicate) P (x1, ..., xn). May be we are searching all n-tuples that meet the criteria.

It can be optimization problems, localization, games, ... They are usually complex problems, in which the only way to find the solution is to analyze (all) the possibilities → exponential growth rate

1 – Introduction

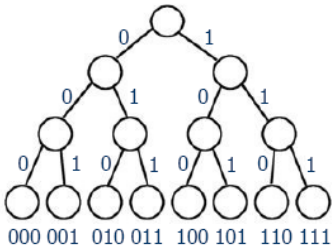
Brute force resolution:

In principle, the problem can be solved by testing all combinations (x1, x2, x3, ..., xn), to see which one optimizes (or meets) P.

The number of candidate n-tuples is: $\prod_{i=1}^{n} |S_i|$

Example: By generating all possible combinations of n bits, we can solve the problem of rucksack 0/1 for n objects.

$T(n) = 2\,T(n-1) + 1 \in O(2^n)$

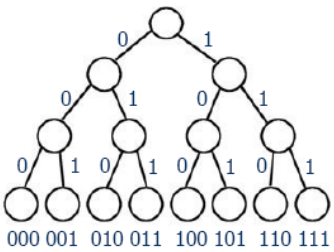


1 – Introduction

Brute force resolution:

Example: By generating all possible combinations of n bits, we can solve the problem of rucksack 0/1 for n objects.

```
method binary_combinations (int[] V, int pos)
  if (pos == V.size)
        process_combination(V);
  else
        V[pos]=0;
        binary_combinations(V,pos+1)
        V[pos]=1
        binary_combinations (V,pos+1)
  end_if
end_method
```



1 – Introduction

Brute force resolution: analysis of the proposal

✓ Implicitly, a tree structure is imposed on the set of possible solutions (solution space).

✓ The way in which solutions are generated is equivalent to go over a tree, whose leaves correspond to possible solutions to the problem.

✓ All options are explored until a solution is found. Analyzing sometimes options that it is already known do not lead to a solution.

1 – Introduction

Brute force resolution: analysis of the proposal

Possible improvement:

➢ Eliminate the need to reach all the leaves of the tree.

➢ If, for an internal node of the tree, we can ensure that we do not reach a solution (that is, it does not lead us to useful leaf nodes), then we can skip that entire branch of the tree, and go backtracking, to move on to explore another branch.

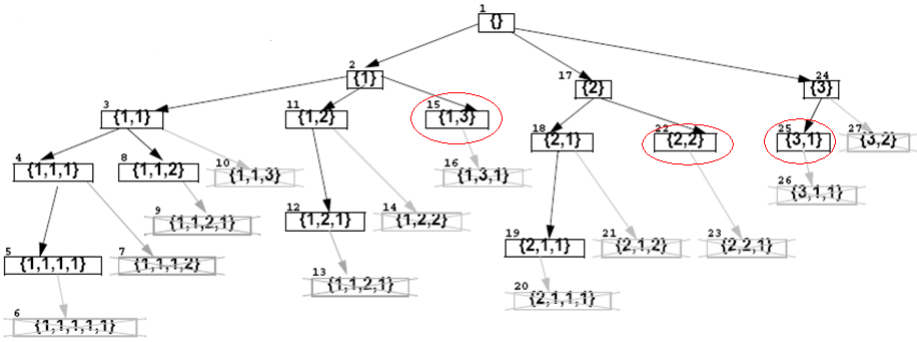
➢ The advantage is that we reach the solution earlier.

1 – Introduction

Informal example of use of the improvement for backtracking

Coins change:

Coin values: {v1 = 1, v2 = 2, v3 =3} Change: 4



Unexplored nodes Solutions

1 – Introduction

Differences with previous approaches:

- In *greedy* algorithms, the solution is built using the possibility of calculating it in chunks: a candidate is never discarded once chosen. With backtracking, however, choosing a candidate in one stage is not irrevocable. In fact, elements are added and removed, until all combinations are tested.

- The type of problems in which we apply backtracking, cannot be broken down into independent sub-problems, so the *divide & conquer* technique is not be applicable.

- In *dynamic programming*, when all the subproblems are solved directly, the direct study of all the cases is actually "Backtracking". In DP the results are stored to avoid recalculation, Backtracking is used when this cannot be done.

1 – Introduction

Specific elements of backtracking algorithms:

Solution: $(x_1, x_2, x_3, \ldots, x_n)$, $x_i \in S_i$.

Solution space of size $\Pi |S_i|$. Backtracking algorithms determine the solutions to the problem by systematically searching the problem space. This search can be represented as a *solution tree*, associated to the problem solution space.

Partial solution: $(x_1, x_2, \ldots, x_k, ?, ?, ..., ?)$, $x_i \in S_i$. Solution vector for which all its components have not yet been established. In backtracking, each tuple is built progressively, element by element, checking that for each element xi added to the tuple, (x1, ..., xi) can lead to a complete satisfactory tuple.

1 – Introduction

Specific elements of backtracking algorithms:

Pruning/limiting function (also called partial objective function, limiting predicate, or feasibility test): P (x1, ..., xi). Function that allows us to identify if a partial solution leads or not to a solution to the whole problem.

✓ Difference between brute force and backtracking: If it is found that $(x_1, \ldots, x_i)$ cannot lead to any solution, it is not necessary to analyze the $|S_{i+1}| \cdot \ldots \cdot |S_n|$ tuples beginning with $(x_1, \ldots, x_i)$.

In order to build the pruning function we must use problem specific restrictions.

1 – Introduction

Specific elements of backtracking algorithms:

Problem specific **restrictions**, which let us to know if a n-tuple is solution:

- *Explicit* restrictions: Limit the possible values of the x_i variables (all the set of values that accomplish these restrictions define the space of solutions of the problem, of size $\Pi |S_i|$).

Example: $x_i \geq 0 \Rightarrow S_i = \{\text{real number no negative}\}$
$x_i \in \{0,1\} \Rightarrow S_i = \{0, 1\}$
$l_i \leq x_i \leq u_i \Rightarrow S_i = \{a: l_i \leq a \leq u_i\}$

- *Implicit* restrictions: Fix the relations between the x_i variables (determining the n-tuples that satisfy the criteria $P(x_1, ..., x_n)$, indicating when the partial solution is able to generate a solution).

1 – Introduction

Solution space terminology:

✓ The solution space (state space) is made up of the set of tuples that satisfy the explicit constraints, and can be structured as an *exploration tree*: at every level the decision of the corresponding step is made.

✓ *State node* (or *problem node*): Each of the nodes of the exploration tree. The path from the root to the node represents a partial or complete tuple, which satisfies the explicit constraints.

✓ *Solution node*: State node for which the path from root to node represents a complete tuple that satisfies the implicit constraints of the problem.

1 – Introduction

Solution space terminology:

- *Live node*: Problem state/node that has been generated but for which all its children nodes have not been generated yet.
- *Dead node* (or *failure node*): Problem state/node that has been generated and, it has been pruned or all its children have been generated.
- *E-node* (expansion node): Live node where a child is being generated now.
- *Promising node*: Node that it is no solution but where it is already possible to generate a solution from its position.

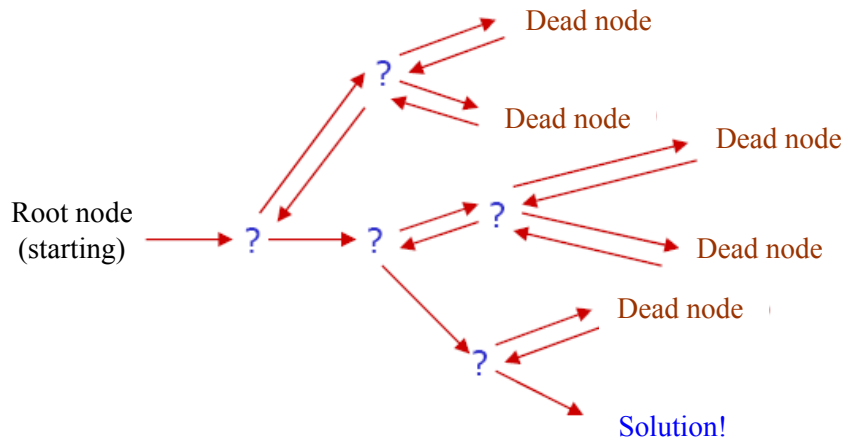
1 – Introduction

State generation mechanism:

- To generate all the states of a problem, we start with a root node, from which the other nodes are generated..
- As we generate problem states, we keep a list of live nodes.
- Pruning functions are used to "kill" live nodes without having to spawn all of their children nodes.
- There are different ways to generate the states of a problem, depending on how we explore the state tree. In the backtracking technique, this exploration is carried out in *depth*:
  - As soon as a new child (N) of the current E-node (A) has been generated, this child becomes a new E-node.
  - A will become E-node again as soon as the subtree N has been fully explored.

1 – Introduction

State generation mechanism:



1 – Introduction

Summary of the characteristics of the backtracking algorithms:

- They are a systematic way of going through the entire solution space.
- Depending on whether we search for just one solution, the optimal one, or all the possible ones, the algorithm stops once the first solution is found, or continues searching for the rest of the solutions. – That means three different approaches --
- These algorithms do not explicitly create or manage the tree; it is implicitly created with *recursive* calls to the algorithm.

2 – General schema

Basic elements of the schema:

A solution can be expressed as an n-tuple (x_1, …, x_n), which satisfies some constraints, and perhaps optimizes an objective function.

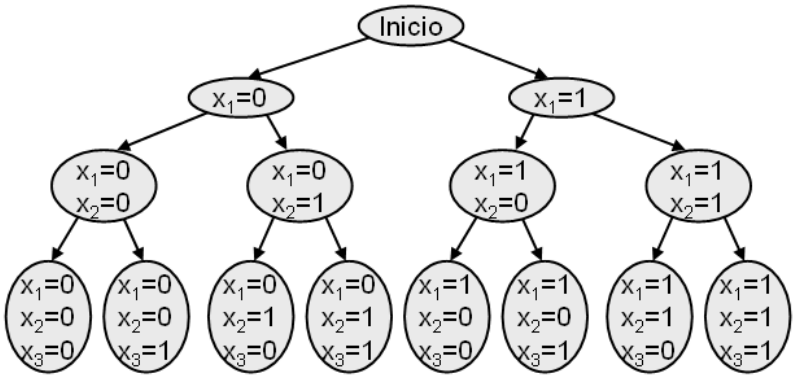
At each moment, the algorithm will be at a certain level k, with a partial solution (x_1, …, x_k). At this point:

- If it is possible to add a new element x_{k+1} to the solution, the new solution is generated and the algorithm advances to the k + 1 level.
- If it is not, other values for x_k are checked.
- If there is no available value to test, then, the algorithm backs to the previous level, k-1.
- It continues until the partial solution is a complete/final solution to the problem, or until there are no more possibilities to test.

2 – General schema

Basic elements of the schema:

The result is equivalent to taking an in-depth search of the solution tree:



2 – General schema

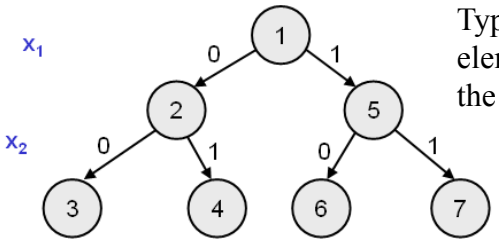
Backtracking trees:

- ✓ The tree is simply a way of representing the execution of the algorithm.
- ✓ It is implicit, not stored (not necessarily).
- ✓ The search is in depth, normally from left to right..
- ✓ The first decision to apply backtracking: what is the tree shape like?
- ✓ Related questions:
  - How is the representation of the problem solution?
  - What the meaning of each value of the solution tuple (x_1, …, x_n)?

2 – General schema

Common types of backtracking trees: binary

$s = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{0, 1\}$



Type of problems: Choose certain elements from a set, regardless of the order of the elements.

Examples:

- The 0/1 rucksack problem.
- Select a subset of elements from a collection fulfilling a condition.

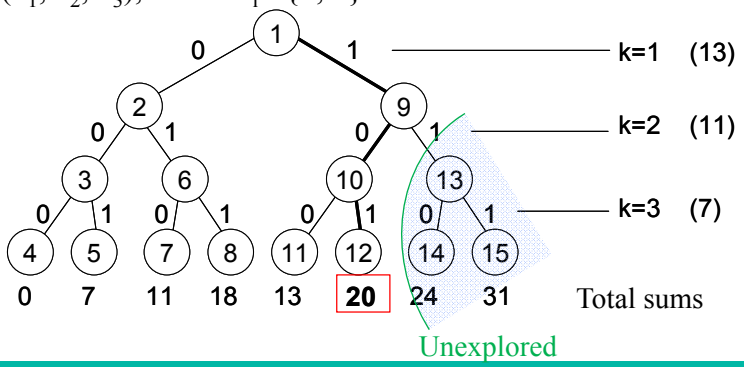
Find a subset of {13, 11, 7} whose sum is exactly 20.

2 – General schema

Common types of backtracking trees: binary, example

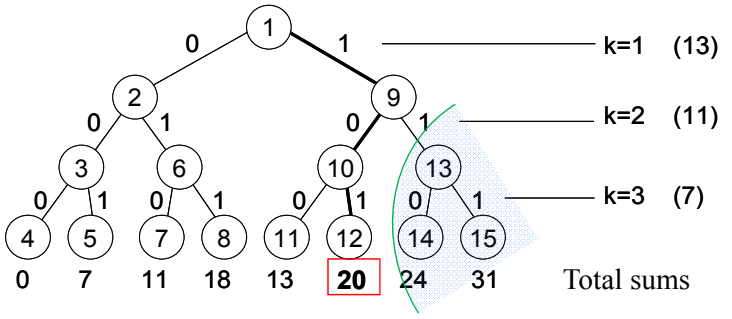
Examples: Find a subset of {13, 11, 7} whose sum is exactly 20.

If it is used a binary tree, at each level i it is decided whether or not element i is part or not of the solution. Representation of the solution: (x_1, x_2, x_3), where $x_i = \{0, 1\}$.



2 – General schema

Common types of backtracking trees: binary, example

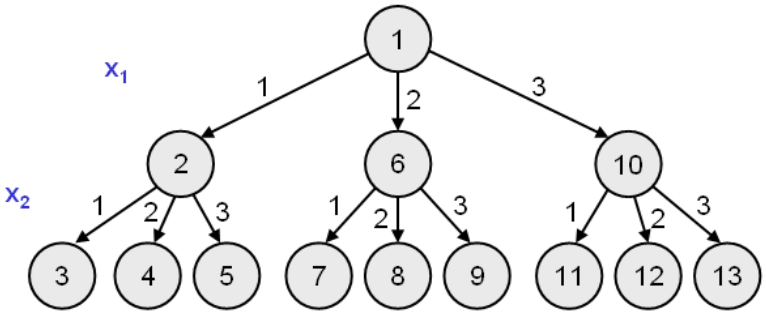


- Each node represents a step of the algorithm, a partial solution at each given moment.
- The tree represents an order of execution (in depth) but it is not stored anywhere.
- A solution is a leaf node with a sum value of 20.
- Improvement: In each node we carry the value of the sum up to that point. If the value is greater than 20: go back to the previous level.

2 – General schema

Common types of backtracking trees: *k-arian* trees

$s = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{1, ..., k\}$



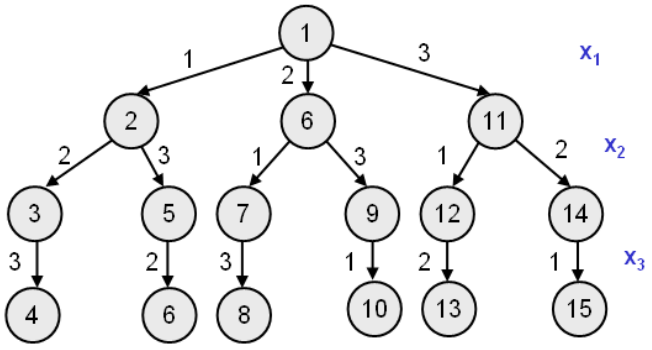
Type of problems: several options for every x_i.

+ Problem of the coin change (no Bellman).
+ Problem of the n queens.

2 – General schema

Common types of backtracking trees : permutational

$s = (x_1, x_2, \ldots, x_n)$, con $x_i \in \{1, ..., n\}$ y $x_i \neq x_j$



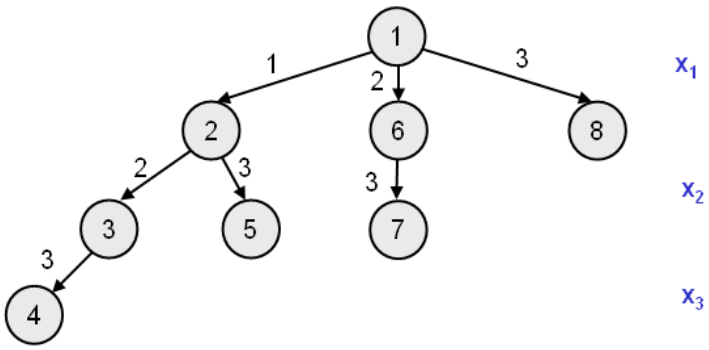
Type of problems: Different x_i can no have same values (no repeated).

- Generate all the permutation of (1, ..., n).
- Assign n activities to n persons, one-to-one assigning.

2 – General schema

Common types of backtracking trees: combinatorial

$s = (x_1, x_2, \ldots, x_m)$, con $m \leq n$, $x_i \in \{1, ..., n\}$ y $x_i < x_{i+1}$



Type of problems: Same as binary trees.

Binary: (0, 1, 0, 1, 0, 0, 1) → Combinatorial: (2, 4, 7)

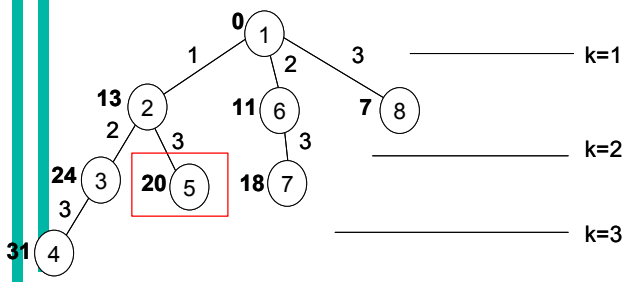
2 – General schema

Common types of backtracking trees: combinatorial

$s = (x_1, x_2, \ldots, x_m)$, with $m \leq n$, $x_i \in \{1, ..., n\}$ and $x_i < x_{i+1}$

Example of correspondence binary - combinatorial

Subset of {13, 11, 7} whose sum is exactly 20.



- At every level i, it is decided which remaining element (1, 2 or 3) is added.
- Solution representation: $(s_1, ..., s_m)$, with $m \leq n$ and $s_i \in \{1, 2, 3\}$.

Every node is a possible solution. It is valid if the sum is 20.

Tree covering is in depth too.

2 – General schema

Questions to solve before developing backtracking solution

- What type of tree is suitable for the problem?
  - ✓How is the representation of the solution?
  - ✓How is the solution tuple? What does each x_i indicate and what values can it store?
- How to go over the nodes according to that tree??
  - ✓Generate a new level.
  - ✓Generate the brothers at a level.
  - ✓Go back in the tree.
- Which branches can be ruled out for not leading to problem solutions?
  - ✓Pruning due to problem restrictions.
  - ✓Pruning according to the objective function criteria.

2 – General schema

Steps building the backtracking algorithm

- ❑ Organize the space of possible solutions in the form of a tree, to facilitate the search.
- ❑ Fix the decomposition in steps.
- ❑ Go over the whole tree:
  - o Solution node → Identify nodes that can be solution.
  - o Problem node → Nodes that can generate solution fragments.
  - o Failure node → Nodes whose descendants are not able to be a solution.
- ❑ Upon finding a failure node, go back to its predecessor..
- ❑ If we are in a problem node, and when exploring it is seen as failure one, go back.
- ❑ If a possible solution node is found it is not, go back.

2 – General schema

General schema (no recursive) for one solution with backtracking

```
Method Backtracking (..): Tuplasolution
        level = 1
        s = s.Initial
        end = false
        repeat
            Generate (level, s)
            if Solution (level, s) then
                        end = true
            else if Criterion (level, s) then
                        level = level + 1
            else while NO MoreBrother(level, s) do
                        Back (level, s)
            endif
        until end
        return s
endmethod
```

Problem of satisfaction of restrictions: we look for any solution that meets certain property (Solution), and it is assumed that there is one.

// Found solution

2 – General schema

General schema (no recursive) for one solution with backtracking

Problem of satisfaction of restrictions: we look for any solution that meets certain property (Solution), and it is assumed that there is one.

Variables:

- ✓ s: Stores the partial solution till the actual state.
- ✓ sINITIAL: Initialization of s.
- ✓ level: Mark the tree level where the algorithm it is.
- ✓ end: It gets the value true when a solution is found.
- ✓ actualValue: temporal variables with actual value (benefit, weight, etc.) of the actual solution.

2 – General schema

General schema (no recursive) for one solution with backtracking

Problem of satisfaction of restrictions: we look for any solution that meets certain property (Solution), and it is assumed that there is one.

Methods:

1.Generate(level, s): It generates the next brother for the same level or the first brother for the next level. It returns the next value to add to the partial solution s (depending of the actual s and level).

2.Solution(level, s): It check if the actual s, (s[1], ..., s[level]) is a solution of the problem.

3.Criterion(level, s): It checks if from (s[1], ..., s[nivel]) it is possible to reach a valid solution. Otherwise all its descents will be avoided (bounding/cutting).

2 – General schema

General schema (no recursive) for one solution with backtracking

Problem of satisfaction of restrictions: we look for any solution that meets certain property (Solution), and it is assumed that there is one.

Methods:

4.MoreBrothers(level, s): It returns true if there are more brothers of the actual node not generated yet.

5.Back(level, s): Go back a level in the solutions tree. Discount 1 from level, probably it must to modify the actual solution, quiting the backtracked elements.

E.D.A. II – Fernando Bienvenido – Curso 2020/21

2 – General schema

General schema (no recursive) for one solution with backtracking

Problem of satisfaction of restrictions: we look for any solution that meets certain property (Solution), and it is assumed that there is one.
Example: obtain a subset of the set of integers T= $\{t_1, t_2, ..., t_n\}$ that added gives exactly P (we know there is a solution).

- Tree: binary

- Variables:

  s: array [1..n] of {-1, 0, 1}
  s[i] = 0 → element i-th not used
  s[i] = 1 → element i-th used
  s[i] = -1→ initialization value (element i-th not studied)
  sInitial: (-1, -1, ..., -1)
  end: It is true when the solution has been find.
  tact: Total cumulated till the moment (initialized to 0).

2 – General schema

General schema (no recursive) for one solution with backtracking

Example: obtain a subset of the set of integers T= $\{t_1, t_2, ..., t_n\}$ that added gives exactly P (we know there is a solution).

- Algorithm: Previous scheme.

- Methods:

  + Generate (level, s)
    s[level] = s[level] + 1
    if s[level]=1 then tact = tact + tlevel

  + Solution (level, s)
    return (level=n) and (tact=P)

  + Criterion (level, s)
    return (level < n) and (tact ≤ P)

E.D.A. II – Fernando Bienvenido – Curso 2020/21

2 – General schema

General schema (no recursive) for one solution with backtracking

Example: obtain a subset of the set of integers T= $\{t_1, t_2, ..., t_n\}$ that added gives exactly P (we know there is a solution).

- Algorithm: Previous scheme.

- Methods:

```
+ MoreBrothers (level, s)
      return s[level] < 1

+ Back (level, s)
      tact = tact – tlevel*s[level]
      s[level] = -1
      level = level – 1
```


2 – General schema

Alternative for the general scheme (no recursive) for one solution using backtracking: it is possible no solution

```
Method Backtracking (..): Tuplasolution
      level = 1
      s = s.Initial
      end = false
      repeat
            Generate (level, s)
            if Solution (level, s) then
                  eng = true                  // Found solution
            else if Criterion (level, s) then
                  level = level + 1
            else while NO MoreBrother(level, s) and (level>0) do
                  Back(level, s)
            fsi
      until end OR (nivel=0)
      return s
fmethod
```

2 – General schema

Alternative for the general scheme (no recursive) for one solution using backtracking: obtaining all the solutions

```
Method Backtracking_all (..): Tuplasolution
        level = 1; s = s.Initial; end = false
        repeat
            Generate (level, s)
            if Solution (level, s) then
                        store(level,s)              // Found a solution
            else if Criterion (level, s) then       // Sometimes intermediate
                        level = level + 1
            else while NO MoreBrother(level, s) and (level>0) do
                        Back (level, s)
            fif
        until (nivel=0)
        return solutions                            // Return all the solutions
fmethod
```


2 – General schema

Alternative for the general scheme (no recursive) for one solution using backtracking: optimization, best value of a function (one)

```
Method Backtracking_os (..): Tuplasolution
        level = 1;  s = s.Initial;   end = false
        voa = -∞; soa = Ø       // voa: actual optimal value;
        repeat                  // soa: actual optimal solution
            Generate (level, s)
            si Solution (level, s) and Value(s) > voa  then
                        voa = Value(s); soa = s
            end_if              //  Sometimes an intermediate solution
            if Criterion (level, s) then     // Sometimes no other level
                        level = level + 1
            else while NO MoreBrother(level, s) and (level>0) do
                        Back (level, s)
            endif
        until (nivel=0)
        return soa              // The best stored solution
fmethod
```

2 – General schema

Alternative for the general scheme (no recursive) for one solution using backtracking: optimization, best value of a function (one)

Problem example: Find a subset, s, of the set T= $\{t_1, t_2, ..., t_n\}$, which elements sum be exactly P, using the minimum number of elements.

- Functions/methods:
  - > Option 1:
    - +Value(s)
      - return s[1] + s[2] + ... + s[n]
    - + All the other remain unchanged.
  - > Option 2: Add variable, vact: number of elements actual tuple.
    - + Initialization, add this: vact = 0
    - + Generate, add this: vact = vact + s[nivel]
    - + Back, add this: vact = vact – s[nivel]

2 – General schema

Recursive general schema of the backtracking approach

✓The backtracking solution is found by adding elements to a partial solution, which is implemented searching a tree.

✓Trees are intrinsically recursive structures, the management of which almost always requires recursion, especially with regard to their search.

✓Simple implementation can be achieved with recursive procedures.

✓ The search is carried out using a *feasibility function* in the tree searching, which tells us if from the point where we are, that is, when adding an element, it is possible to find the solution (it does not guarantee it).

2 – General schema

Recursive general schema of the backtracking approach

✓The search is carried out using a feasibility function in the tree searching, which tells us if from the point where we are, that is, when adding an element, it is possible to find the solution (it does not guarantee it).

- Any combination with a non-feasible item becomes an infeasible combination.

- When an infeasible element is found, all the combinations that contain it are eliminated, that is, when a failure node is reached, the last decision made must be undone to opt for the next alternative.

2 – General schema

Recursive general schema of the backtracking approach

```
Method Backtracking_R (step): solution;
    InitializeOptions; success =FALSE
    repeat
            SelectNewOption;
            if Acceptable then
                AnotateOption;          // Option is added to the solution
                if IncompleteSolution then
                        Backtracking_R(next_step)
                        if NO success then
                           CancelAnotation
                        endif
                else                    // Complete solution
                        success:=TRUE
                endif
            endif
    until (success) OR (LastOption)
    return solution
endmethod.
```

Obtaining ONE solution

2 – General schema

Recursive general schema of the backtracking approach

Elements of the solution:

Obtaining ONE solution

+ Children generation, where for each node we generate their descendants with the possibility of a solution. This step is called expansion, branching, or bifurcation. (Repeat loop)

+ + For each of these descendants, we must apply what we call failure proof (Acceptable). We check that with what we have we can still reach the solution.

+ + If this node is acceptable, we will apply the solution test that checks whether the node, that is a possible solution, is indeed a complete solution. (No IncompleteSolution).

Backtracking is done thanks to the recursion process (exploring in depth), and the process of annotating and canceling this records (forward and backward).

2 – General schema

Recursive general schema of the backtracking approach

Obtaining ALL the solutions

```
Method Backtracking_Rall (step): solutions[];
 InitializeOptions; success =FALSE
    repeat
          SelectNewOption;
          if Acceptable then
           AnotateOption;            // Option is added to the solution
              if IncompleteSolution then
                   Backtracking_Rall(next_step)
                   if NO success then
                      CancelAnotation
                   endif
              else                   // Solution is included to solution list
                   AddSolution
              endif
          endif
    until (UltimaOpcion)
    return solutions
endmethod
```

Home task: Generate equivalent recursive schema for the optimization case

2 – General schema

Conclusions about the general backtracking schema

+ Backtracking: Comprehensive and systematic exploration of a solution tree.

+ Applications steps:
  - Decide the type of solution tree.
  - Establish the algorithm schema.
  - Design specifically the functions/methods of the schema.

+ It is relatively easy to design algorithms that find optimal solutions but .....
  - Backtracking algorithms are very inefficient.

Improvements: improve pruning mechanisms and include other types of routes (not only in depth) → Branch & Bound approach.

3 – Analysis of execution time

General considerations:

+ The efficiency of these algorithms depends on:

  ❑ The number of nodes to be explored for each case, which is impossible to calculate exactly a priori.
  ❑ The time required to process each node, which is given by the cost of the functions.

+ Lets n to be the number of stages necessary to build the solution (that is, the maximum length of the solution tuple, or the depth of the exploration tree) and [0..v-1] the range of values that can be assigned to every step decision. Example: treatment and use of a k-arian tree.

3 – Analysis of execution time

General considerations, k-arian tree:

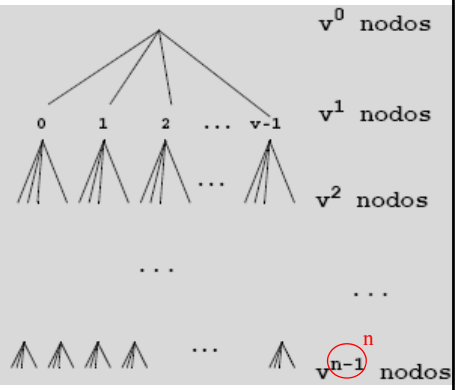
+ Best case: solution is found in the first in depth search. The algorithm musts generate and analyze n nodes:

O(n) - Pending of the evaluation of the methods.

+ Worst case: it is required to seach the whole tree. It is possible to evaluate the upper bound (pending of the evaluation of the methods):

$$nodes = \sum_{i=0}^{n} v^i \approx v^n$$

$O(v^n)$ upper bound



3 – Analysis of execution time

Example with binary or combinatorial trees:

Example of the problem of choosing elements of a subset (selection of values that add up a value, backpack 0/1, ...). Solution binary trees, the number of nodes to consider is:

$$numNodes(n) = 2 + 2^2 + 2^3 + ... + 2^n = 2^{n+1} - 2$$ ** (check)

Similarly, if combinatorial type trees are used, it can be seen that the number of nodes is:

numNodes(n) = n! (really n!-1)

In the worst case, the number of nodes is 2^n or n!; in this case the execution time will be of order O ($p(n)2^n$) or O ($q(n)n!$), respectively, with p and q polynomials in n (representing cost of internal methods).

3 – Analysis of execution time

Analysis of the applicability of the Backtracking approach:

+ Generally, backtracking algorithms result in exponential or factorial time orders → Do not use if there are other faster alternatives.

+ The use of restrictions, both implicit and explicit (which allow declaring a node as failed, exploring no longer), tries to reduce this time, but in many cases it is not enough to achieve treatable algorithms, whose execution times are of reasonable complexity order.

+ When looking for just a solution (not all or an optimum). We can consider the possibility of different ways of generating the tree nodes, to achieve this we only have to vary the order in which the descendants of a node are generated, so that it tries to generate the most appropriate for our strategy.

3 – Analysis of execution time

Analysis of the applicability of the Backtracking approach:

+ The feasibility/check test allows us to see the type of each node is ... and therefore if we can abandon the exploration at that point or not.

+ Check tests will determine the efficiency of the backtracking algorithms.

Memory efficiency:

Backtracking algorithms have specific memory requirements, typical of the backtracking scheme, O (n), given by the maximum depth of recursive calls.

3 – Analysis of execution time

Recommendations for a good efficiency (relative):

1. Good choice of the set of possible solutions → the shortest possible.
2. Trying to apply the solution test before reaching a solution node → detect as soon as possible if we are in a failed node, so we avoid exploring its children as it is useless work.
3. The test should be applicable with the minimum information.
4. The test must be light so that we do not need extra time, which would be a complication if it does not detect many failure nodes.
5. General rule: try simple ones; sophisticated tests for desperate situations with huge trees.

4 – Examples: Exhaustive search in graphs

Problem approach:

+ Given a connected graph. It starts from a given node and the vertices of the graph are visited in an orderly and systematic way, passing from one vertex to another through the edges of the graph.

  It is used when we want to do some operation with all the nodes of the tree (in our case we will say that it has been visited) or locate the node that optimizes a certain condition (an index is added).

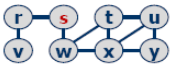
+ Search types:

  o Depth First search: Equivalent to a preorder tree search. This means applying the backtracking strategy.

  o Breadth First Search: Equivalent to a search in the tree by levels. This is a simple branch (& bound) schema.

4 – Examples: Exhaustive search in graphs

In depth search: example



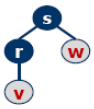
Stack S={}

Stack S={s}

Stack S={r,s}

At the stack S, it is stored the active path (starting by s).

Accessible nodes come out after analyzing edges connecting the last selected node with unselected nodes.



We need to store the set of visited and unvisited nodes.

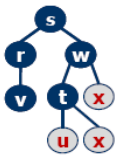
4 – Examples: Exhaustive search in graphs

In depth search: example

Stack S={v,r,s}

Stack S={w,s}

Stack S={t,w,s}



Backtracking is implicit in returning to unvisited nodes when a thread ends.

Stack S ends up giving the route that takes us to the last node to visit.

E.D.A. II – Fernando Bienvenido – Curso 2020/21

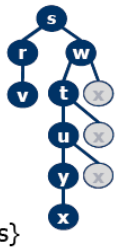
4 – Examples: Exhaustive search in graphs

In depth search: example

Stack S={u,t,w,s}

Stack S={y,u,t,w,s}

Stack S={x,y,u,t,w,s}



Stack S ends up giving the route that takes us to the last node to visit.

The important thing is to visit all the nodes ... that is the true objective of the algorithm.

The route is not unique: it depends on the starting vertex and the order of visit of adjacent vertices.

The order of visit of some nodes can be interpreted as a tree: the depth spanning tree associated with the graph (see example).

4 – Examples: Exhaustive search in graphs

Algorithmic schema of the solution:

```
method DFS_Launcher (Graph G(V,E))        // DFS Depth-First-Search
        for (i=0; i<V.length; i++)
                visited[i] = false
        endfor
        for (i=0; i<V.length; i++)
                if (NO visited[i])
                        DFS(G,i)
                endif
        endfor
endmethod
```

```
Method DFS (Graph G(V,E), int i)
        visited[i] = true;
        for every (v[j] adjacent to v[i])
                if (NO visited[j])
                        DFS(G,j)
                endif
        endForEvery
endmethod
```

Efficiency is O(|V|+|E|) if it is used the graph representation based on adjacent tables

4 – Examples: Exhaustive search in graphs

Working procedure:

✓ It starts by visiting any node.
✓ All connected components hanging from each successor are checke in depth (the paths are examined until nodes already visited or without successors are reached).
✓If after having visited all the descendants of the first node (itself, its successors, the successors of its successors, ...) there are still nodes to visit, the process is repeated from any of these unvisited nodes.

Applications of this general problem (variations):

❑ Analyze the robustness of a computer network represented by a graph.

❑ Examine whether a directed graph has cycles, before applying any algorithm that requires it to be acyclic. (Modification is required).

4.2 – Example: The 0/1 Rucksack problem

Problem statement:

+ Data: n: number of available elements/objects.
W: size (weight) of the rucksack.
$w = (w_1, w_2, ..., w_n)$ weight of the elements/objects.
$b = (b_1, b_2, ..., b_n)$ benefit of the elements/objets.

+ Solution: $\{x_1,..,x_i ,\ldots,x_n\} / x_i \in \{0,1\}$.

+ Objective: Select the objects/elements that maximize the benefit, but with a cumulated weight smaller or equal toW.

Maximize $(\Sigma x_i \cdot b_i) / \Sigma x_i \cdot w_i \leq W$ with $i=1..n$, $x_i \in \{0, 1\}$.

4.2 – Example: The 0/1 Rucksack problem

Using the backtracking schema (methodic approach):

1. Determine what the type of the backtracking tree ↔ how the representation of the solution is like.
2. Choose the appropriate algorithm scheme, adapting it if necessary.
3. Design an specific version of the generic functions for the given application: according to the shape of the tree and the problem characteristics.
4. Possible improvements: use local variables with accumulated values, prune the tree, etc.

4.2 – Example: The 0/1 Rucksack problem

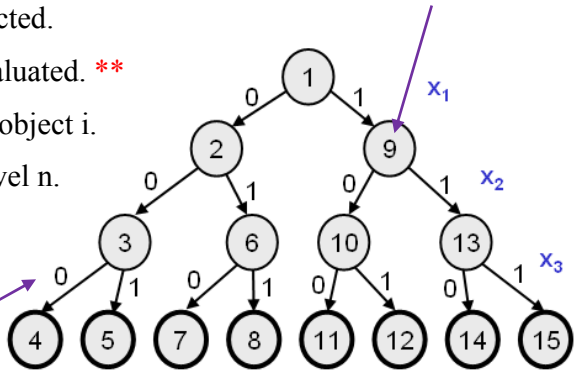
Representation of the solution: (1) Binary tree

s= $(x_1, x_2, ..., x_n)$, with $x_i \in \{0,1\}$ **

- $x_i = 0 \rightarrow$ Object i is NO selected.
- $x_i = 1 \rightarrow$ Object i IS selected.
- $x_i = -1 \rightarrow$ Object i no evaluated. **
- At level i, it is evaluated object i.
- Solutions nodes are at level n.

Every level represents an object.
(0 no selected, 1-selected)

Numbers represent the order of checking the nodes.



Nodes store the cumulated weigth.

4.2 – Example: The 0/1 Rucksack problem

Representation of the solution: (2) Combinatorial tree

$s = (x_1, x_2, ..., x_m) / m \leq n, x_i \in \{1, ..., n\}$ and $x_i < x_{i+1}$

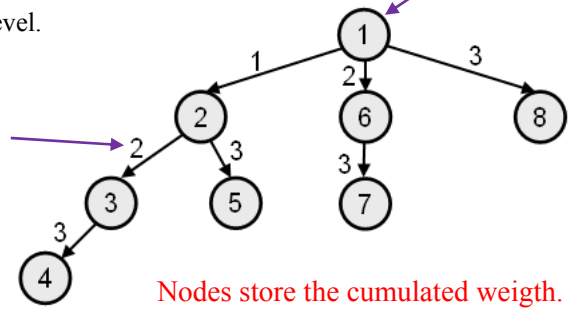
$x_i \rightarrow$ Number of the selected object.

$m \rightarrow$ Total number of selected objects.

Solutions are at every level.

The edge number represents that the object with that number is selected.

Numbers represent the order of checking the nodes.



Nodes store the cumulated weigth.

4.2 – Example: The 0/1 Rucksack problem

Algorithmic schema: iterative / optimization

```
Method Backtracking_RS (real[] b,w): Tuplesolution
        level = 1;  s = s.Inicial;
        voa = -∞; soa = Ø        // voa, soa: actual value and solution;
        wact=0; bact=0           // actual weight and benefit
        repeat
            Generate (level, s)
            if Solution(level, s) AND (bact > voa)  then
                        voa = Value(s); soa = s
            else if Criterion (level, s) then
                        level = level + 1
            else
                    while NO MoreBrother(nivel, s) AND (level>0)  do
                        Back (level, s)
            endif
        until (level=0)
        return soa                  // Best solution stored
endmethod
```

Binary tree

4.2 – Example: The 0/1 Rucksack problem

Algorithmic schema: optimization

- In order to calculate the weight and benefit at every node, it is possible to use local variables wact, bact, which store the cumulated weight and benefit.
- The solution array is s: array [1..n] of –1,0,1.
  - ✓s[i] = 1, 0. Means that the object i is included or not.
  - ✓s[i] = -1. No evaluated the object i (initialization value).
- Being an optimization problem, it is require to check all the node to assure the solution. Finishing when level == 0 (returning to root node).
- During the execution we have the best solution till actual node. If a new solution is found, check whether it is better than the current one.
- Variable voa stores the value of the best solution up to the actual node and soa the objects that compose.

4.2 – Example: The 0/1 Rucksack problem

Generic functions/methods of the schema:

- Generate (level, s) → Test first 0, later 1

```
s[level]= s[level] + 1

wact= wact + p[level]*s[level]
bact= bact + b[level]*s[vel]

if s[level]=1 then
  wact= wact + w[level]
  bact= bact + b[level]
endif
```

- Solution (level, s) → Check the leave nodes that fulfill the rucksack weight restriction.

```
return (level=n) AND (wact ≤ W)
```

4.2 – Example: The 0/1 Rucksack problem

Generic functions/methods of the schema:

- Criterion (level, s) → It shows if the weight restriction is fulfilled and not at the bottom level.

```
return (level<n) AND (wact ≤ W)
```

- MoreBrother (level, s)

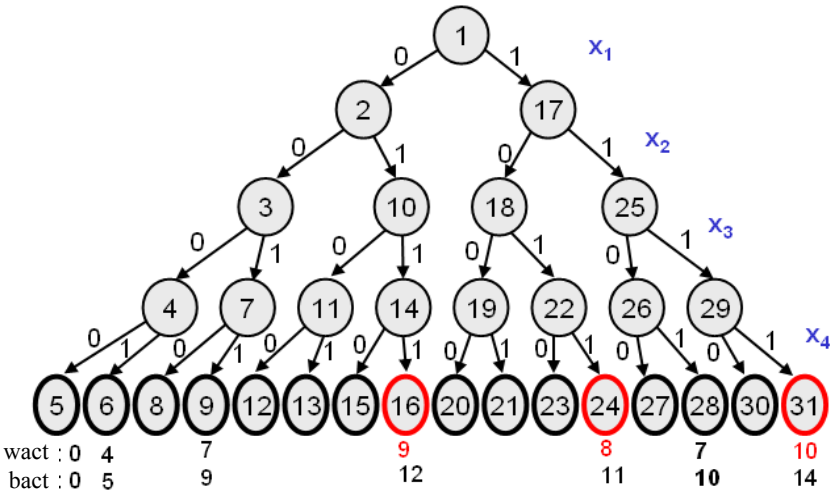
```
return s[level] < 1
```

- Back (level, s)

```
wact= wact – w[level]*s[level]
bact= bact – b[level]*s[level]
s[level]= -1
level= level – 1
```


4.2 – Example: The 0/1 Rucksack problem

Example: n = 4; W = 7; b = (2, 3, 4, 5); w = (1, 2, 3, 4)



4.2 – Example: The 0/1 Rucksack problem

UAL

Efficiency analysis of the solution:

✓ This algorithms solves the problem, finding the optimal solution, but in an inefficient way.

✓ Complexity order of the algorithm:
Number of generated nodes= 2^{n+1}-1. Order $O(2^n)$.

✓ Additional problem: for this example, all the possible nodes are generated, no pruning function used. The Criterion function is always true with the only exception of some leave nodes.

✓ Solution: Improve the pruning using a more restrictive Criterion function/method.

4.2 – Example: The 0/1 Rucksack problem

UAL

Improving the method Criterion with bounding conditinos:

+ It is possible to prune according to the optimization criteria:

- ❑ Prune according to the weight criterion: if the current weight is greater than W, prune the node.
- ❑ Prune according to the optimization criterion: if the current benefit cannot improve the voa, prune the node.

+ Despite this pruning, in the worst case, the order of complexity remains O $O(2^n)$.

+ On average, pruning is expected to remove many nodes, reducing total time, but execution time remains too high.

4.2 – Example: The 0/1 Rucksack problem

Structural improvements:

+ Use the concept of density of value:

  ❑ For each node, first generate the value 1 and then the value 0 (instead of first 0 and then 1). Idea: hopefully the solution for rucksack 0/1 will be "similar" to that for rucksack no 0-1.
  ❑ Ordering the object by its b_i/w_i values, then first solution will have value1 at the first positions and good total benefit (biggest value density), improving the pruning because no possible to surpass the voa in later nodes.

+ Using a combinatorial tree.

4.3 – Example: The Salesman problem

Problem statement:

➢ It is sought to determine all the Hamiltonian cycles of a connected graph G = <V, A> with n nodes/vertices.

➢ A Hamiltonian cycle is a path that goes through the n vertices of the graph, visiting each vertex once, until ending at the starting vertex.

➢ Solution array: $(x_1, ..., x_n)$, such that x_i represents the i^{th} vertex visited for the proposed cycle.

➢ All that is needed is to determine the set of possible vertices x_k once the previous $x_1, ..., x_{k-1}$ are selected.

4.3 – Example: The Salesman problem

Implementation conditions:

- If k = 1, x [1] can be any of the n vertices, although to avoid finding the same cycle n times, we will demand that x [1] = 1.
- If 1 <k <n, then x [k] can be any vertex v that is different from vertices x [1], x [2], ..., x [k-1] and it is connected by an edge with x[k-1].
- x [n] can only be the only remaining vertex and must be connected to both x [n-1] and x [1].

- It is used a permutational tree (2).
- The criterion method evaluates the availability of the edges to continue (2) (3).

4.3 – Example: The Salesman problem

Algorithmic schema: methods nextVvalue and hamiltonian

```
Method nextValue (k): int                // Next valid node, 0 in none
// x[1..k-1] nodes already assigned, G[ ] [ ] graph adjacency graph
   x[k] = 0     // Initialization
   do
         x[k]++;
         if (G[x[k-1]][x[k]] && x[k] ∉ x[1..k-1] &&
                              (k<N || (k==N && G [x[k]] [x[1]] ))) then
                  return x[k];
   while (x[k]<N);
   x[k]=0
   return 0
endmethod
```

4.3 – Example: The Salesman problem

Algorithmic schema: methods nextVvalue and hamiltonian

```
method hamiltonian (k)  // x[1..k-1] nodes already assigned
    if (k==N) then
        store x[1..N]    // One solution – Read comments
    else
        do
            x[k] = nextValue(k);
            if (x[k]!=0)
                hamiltonian(k+1);
        while (x[k]!=0);
    endif}
endmethod
```


4.3 – Example: The Salesman problem

Solution comments:

- This algorithm finds "all" the Hamiltonian cycles, we would have to evaluate which one has the minimum cost. In order to do this, we substiture "store solution" by "check the cost against the previously stored, changing it if the actual value is smaller. It is required a variable "cost", initialized to $+\infty$.

- If our objective is the optimum, it is possible to improve the criterion taking account of cumulated cost of the partial solution against the last solution stored (pruning).

- A very efficient greedy heuristic, for this problems, was available, but sub-optimal.

- There are too a dynamic programming solution and a brute force solutions available.

4.4 – Example: Task planning

Problem statement:

Let's consider this situation:

- ✓ There are n persons/workers and n tasks.
- ✓ Every person i is able to complete a task j with an specific performance: B[i, j].
- ✓ Objective: assign a task to every worker (as a one-2-one assignment), maximizing the total performance.

Tasks

Workers	B	1	2	3
	1	4	9	1
	2	7	2	3
	3	6	3	5

NP Complet problem

Example 1 : (P1, T1), (P2, T3), (P3, T2) B_{TOTAL}= 4+3+3 = 10

Example 2 : (P1, T2), (P2, T1), (P3, T3) B_{TOTAL}= 9+7+5 = 21

4.4 – Example: Task planning

Solution statement:

- Problem data:
  - o n: number of persons and tasks available.
  - o B array [1..n, 1..n] of integers, containing the performance or benefit of every possible assignation: B[i, j] = benefit of assigning the person i the task j.

- Result:

  Carry out n assignments $\{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}$.

  ** to discuss **

Mathematical formulation:

Maximize $\sum_{i=1..n} B[p_i, t_i]$, under these restrictions $p_i \neq p_j$, $t_i \neq t_j$, $\forall\ i \neq j$

4.4 – Example: Task planning

Solution statement: representation

1. Assignation pairs:

$s = \{(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)\}$, con $p_i \neq p_j$, $t_i \neq t_j$, $\forall\ i \neq j$

- Task t_i is assigned to person p_i.

- Very wide tree (k-arian).

- It is required to assure a lot of restrictions (no repeating elements).

- No very good representation.

4.4 – Example: Task planning

Solution statement: representation

2. Matrix of assignations:

$s = ((a_{11}, a_{12}, ..., a_{1n}), (a_{21}, a_{22}, ..., a_{2n}), ..., (a_{n1}, a_{n2}, ..., a_{nn}))$ / $a_{ij} \in \{0,1\}$

with $\sum_{i=1..n} a_{ij} = 1$. $\sum_{j=1..n} a_{ij} = 1$.

Representation:

$a_{ij} = 1 \rightarrow$ task j is assigned to person i.
$a_{ij} = 0 \rightarrow$ task j is not assigned to person i.

Tasks

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

Persons

- Binary tree, very deep: a n^2 levels tree.
- Requiring a lot of restrictions too.

4.4 – Example: Task planning

Solution statement: representation

3. Vector of assignation from the point of view of the persons:

   $s = (t_1, t_2, ..., t_n)$, being $t_i \in \{1, ..., n\}$, with $t_i \neq t_j$, $\forall$ $i \neq j$

   - t_i is the number of the task assigned to the person i.
   - This representation generates a permutational tree.

4. Vector assignations from the point of view of the tasks:

   $s = (p_1, p_2, ..., p_n)$, being $p_i \in \{1, ..., n\}$, with $p_i \neq p_j$, $\forall$ $i \neq j$

   - p_i is the number of the person assigned to task i.
   - Inverse representation (dual) to option 3.

4.4 – Example: Task planning

Algorithmic schema: optimization

```
Method Backtracking_os (..): Tuplasolution
        level = 1;  s = s.Initial; actb=0  //actb: actual benefit
        voa = -∞; soa = Ø          // voa: actual optimal value;
        repeat                     // soa: actual optimal solution
            Generate (level, s) // Actualize actb
            if Solution (level, s) and actb > voa  then
                        voa = actb; soa = s
            end_if                 //  Sometimes an intermediate solution
            if Criterion (level, s) then     // Sometimes no other level
                        level = level + 1
            else while NO MoreBrother(level, s) and (level>0) do
                        Back (level, s)
            endif
        until (nivel=0)
        return soa                 // The best stored solution
fmethod
```

4.4 – Example: Task planning

Basic elements of the schema:

+ Variables:
  - s: array [1..n] of integer: every s[i] stores the task assigned to the person i. Initialized to 0 (no task already).
  - actb: cumulated benefit of the actual solution.

+ Generate (level, s) → Test first 1, then 2, ..., n.

```
s[level]= s[level] + 1
if s[level]=1 then
        actb= actb + B[level, s[level]]
else
        actb= actb + B[level, s[level]] – B[level, s[level]-1]
endif
```


4.4 – Example: Task planning

Basic elements of the schema:

+ Criterion (level, s)

```
for i= 1, ..., level-1 do
        if s[level] = s[i] then return false
endfor
return true
```

+ Solution(level, s)

```
return ((level==n) AND Criterion (level, s))
```

+ MoreBrother(level, s)

```
return (s[level] < n)
```

+ Back (level, s)

```
actb= actb – B[level, s[level]]
s[level]= 0
level= level – 1
```

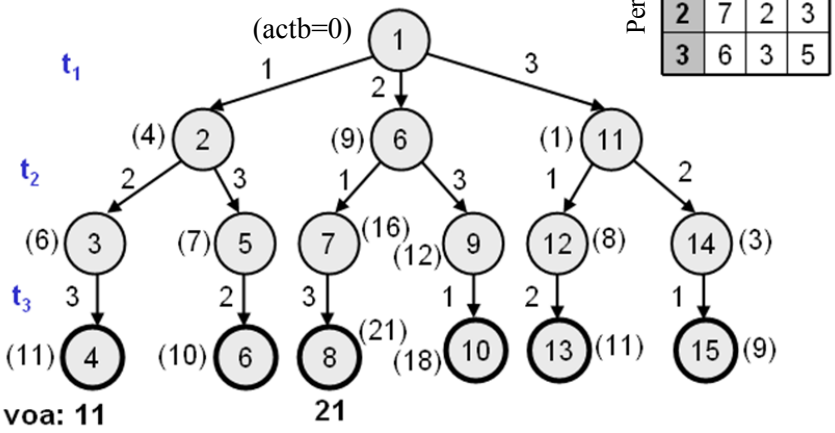
E.D.A. II – Fernando Bienvenido – Curso 2020/21

4.4 – Example: Task planning

Application example: n =3

Tasks

Persons B	1	2	3
1	4	9	1
2	7	2	3
3	6	3	5



4.4 – Example: Task planning

Solution analysis:

+ Problem: function Criterion is very slow, repeating so many checkups.

+ Solution: using an array indicating those task that have been already assigned in the actual state.

✓ used: array [0..n] of integer.
✓ used[i] points the number of times the task i is assigned in the actual planning state (that is, included in s).

Initialization: used[i] = 0, for every i.

4.4 – Example: Task planning

Análisis de la solución: cambios para mejora criterio

```
+ Criterion (level, s)
        return used[s[level]]=1

+ Back (level, s)
        act= actb – B[level, s[level]]
        used[s[level]]--
        s[level]= 0
        level= level – 1

+ Generate (level, s)
        used[s[level]]--
        s[level]= s[level] + 1
        used[s[level]]++
        if s[level]=1 then actb= actb + B[level, s[level]]
        else actb= actb + B[level, s[level]] – B[level, s[level]-1]   endif
```

Los métodos Solución y MasHermanos no cambian.

4.4 – Example: Task planning

Conclusions:

- Efficiency is improved (check how it changes between version 1 and 2, home work).

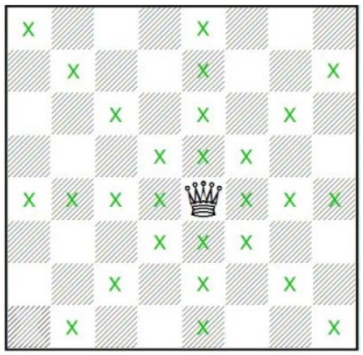
- Anyway it remains inneficience.

- It is possible to improve actual solution, pruning the branches using the optimization criteria… no been required to arrive to every leaf of every branch.

4.5 – Example: Other examples

The n queens problem: origin 8 queens chess problem

This problem involves placing eight queens on a chess board, in such a way that none threatens (can eat) another. A queen threatens another piece that is in the same file, row or either of the two diagonals.



Brute force $\binom{64}{8} = 4.426.165.368$

As there cannot be more than one queen per row, we can reframe the problem: Place a queen in each row of the board so that it is not threatened. In this case, to see if two queens are threatened, just we need check if they share a column or diagonal.

4.5 – Example: Other examples

The n queens problem: origin 8 queens chess problem

- Since each queen must be in a different row, without loss of generality we can assume that queen i is placed in row i.
- All the solutions to this problem can be represented as 8-tuples (x1, ..., x8) in which xi indicates the column in which queen i is placed.
- The problem verifies the necessary conditions so that it can be solved using Backtrack:
  + The solution can be expressed as a n-tuple.
  + The x_i components are selected one by one from a finite set of options (the columns of the table).

E.D.A. II – Fernando Bienvenido – Curso 2020/21

4.5 – Example: Other examples

The n queens problem: origin 8 queens chess problem

+ Solution space:

Size $|S_i|^8 = 8^8 = 2^{24} = 16M$

+ *Explicit* restrictions:

$S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$

(This way possible permutations are limited, size (8!) = 40320)

+ *Implicit* restrictions:

- Not a single pair (x_i, x_j) with $x_i = x_j$ is allowed (all the queens must be located in different columns).
- Not a single pair (x_i, x_j) with $|j-i| = |x_j-x_i|$ is allowed (all the queens must be located in different diagonals).

4.5 – Example: Other examples

The n queens problem: origin 8 queens chess problem

- Applying the restrictions, in each stage k we are going to generate the k-tuples with the possibility of being a solution or part of it.

- The nodes of the tree of depth k in the n-tuple that is being built, if the restrictions are verified, it is a k-promising. If they are not verified, it is a node failure.

- This way, we will generate the tree.

- Efficiency:
  - > Number of possible permutations : 8!
  - > Restrictions testing at level k: O(k)

E.D.A. II – Fernando Bienvenido – Curso 2020/21

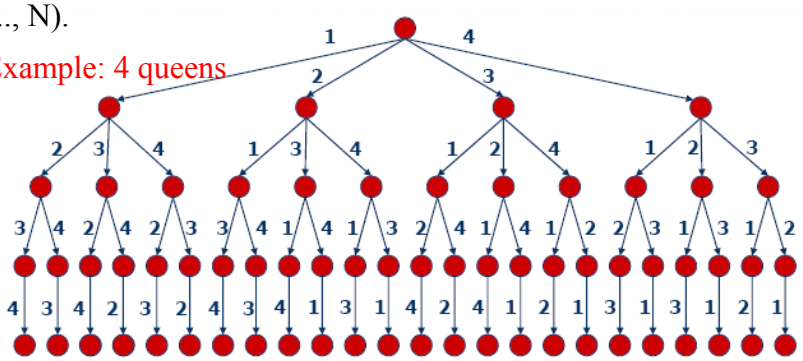
4.5 – Example: Other examples

The n queens problem: generalization of the 8 queens case

Problem formulation: Locate N queens in a NxN boards, in a way they do not attack between them.

- The solution space is composed by N! permutations of the N-tuple (1, 2, ..., N).

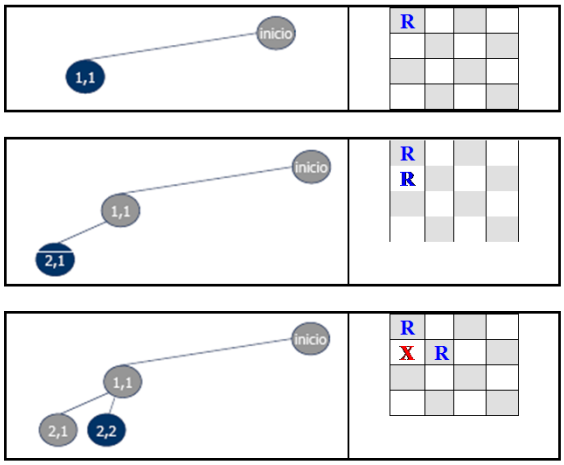
Example: 4 queens



$T(n) \in \theta(n \cdot n!) = \theta(n!) \rightarrow$ NP-completo

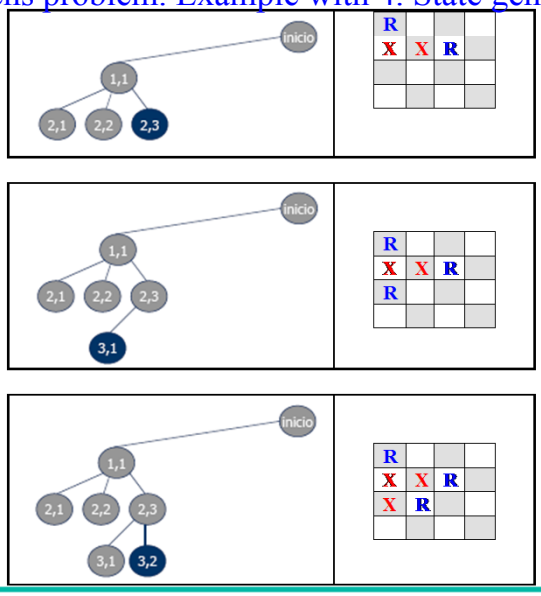
4.5 – Example: Other examples

The n queens problem. Example with 4. State generation.



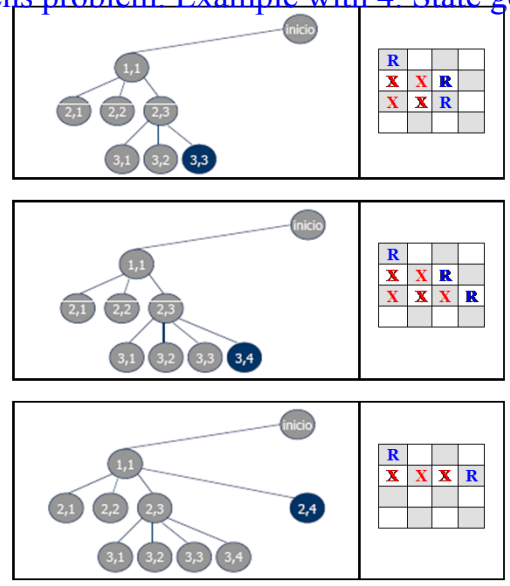
4.5 – Example: Other examples

The n queens problem. Example with 4. State generation.



4.5 – Example: Other examples

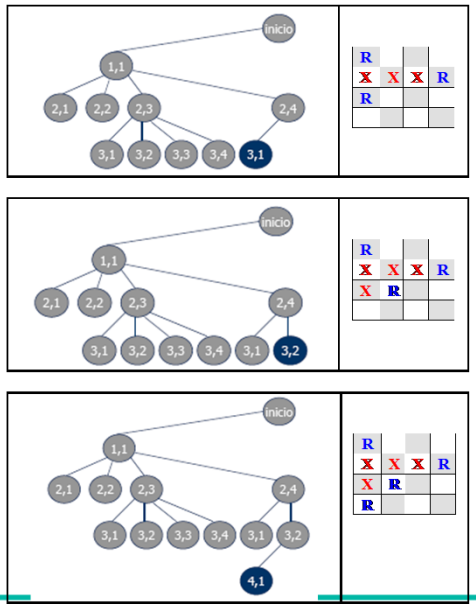
The n queens problem. Example with 4. State generation.



E.D.A. II – Fernando Bienvenido – Curso 2020/21

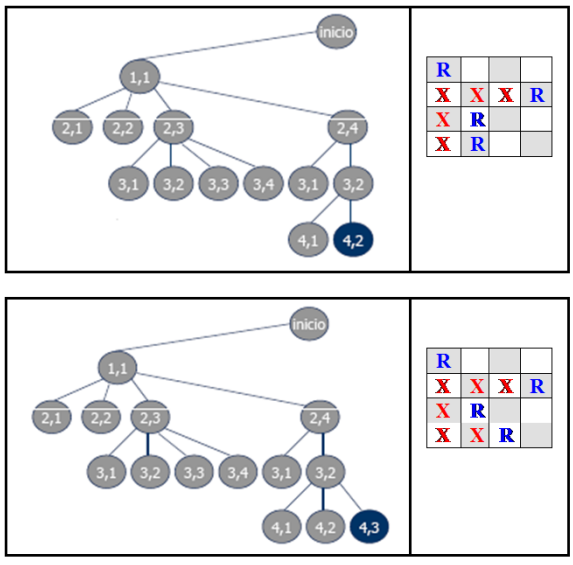
4.5 – Example: Other examples

The n queens problem. Example with 4. State generation.



4.5 – Example: Other examples

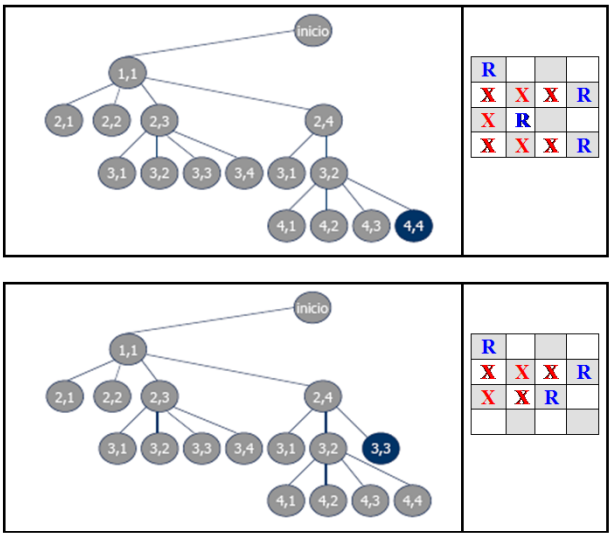
The n queens problem. Example with 4. State generation.



E.D.A. II – Fernando Bienvenido – Curso 2020/21

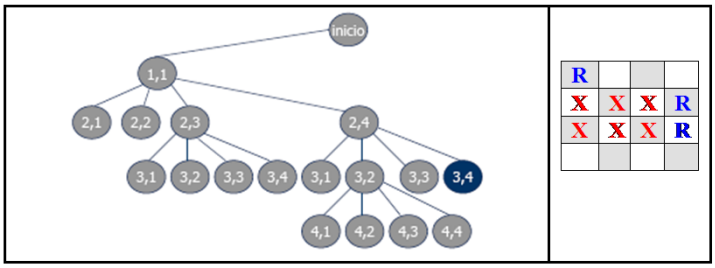
4.5 – Example: Other examples

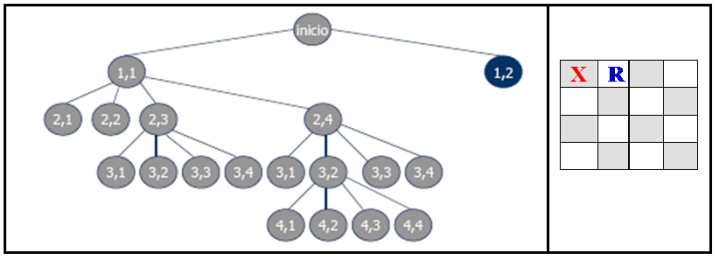
The n queens problem. Example with 4. State generation.



4.5 – Example: Other examples

The n queens problem. Example with 4. State generation.

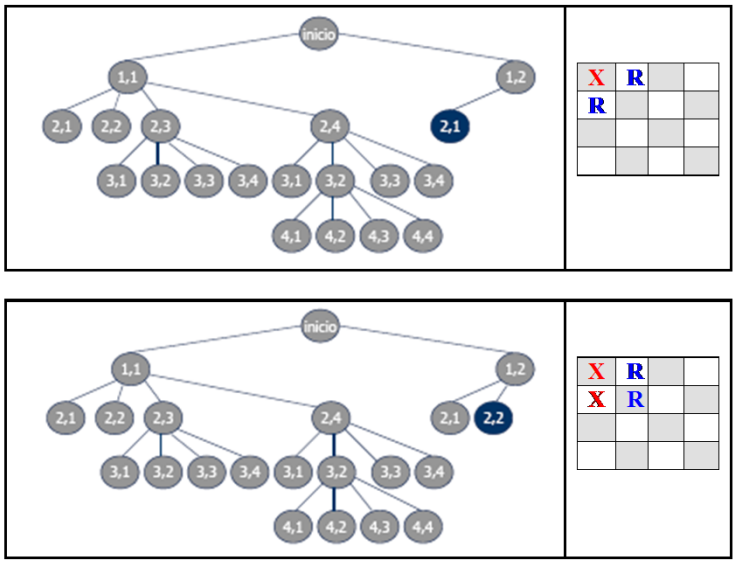




E.D.A. II – Fernando Bienvenido – Curso 2020/21

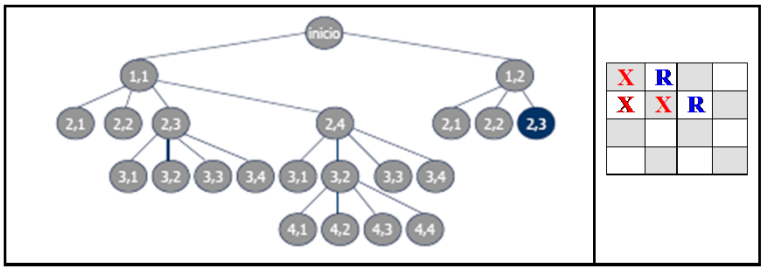
4.5 – Example: Other examples

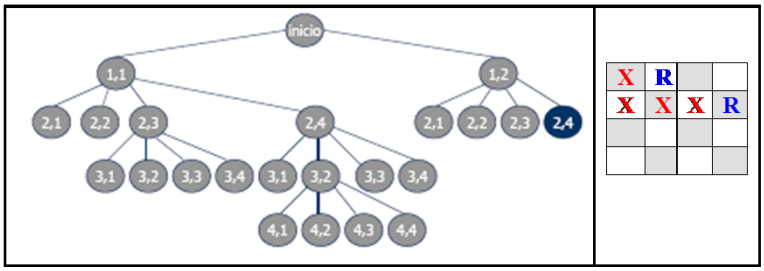
The n queens problem. Example with 4. State generation.



4.5 – Example: Other examples

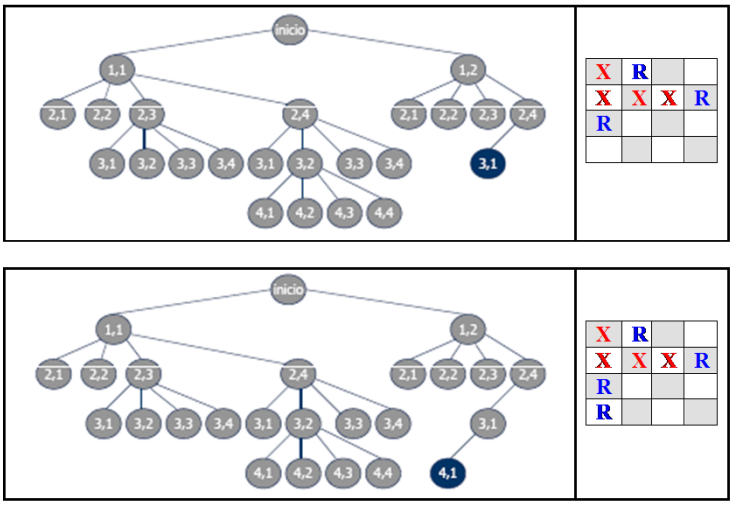
The n queens problem. Example with 4. State generation.





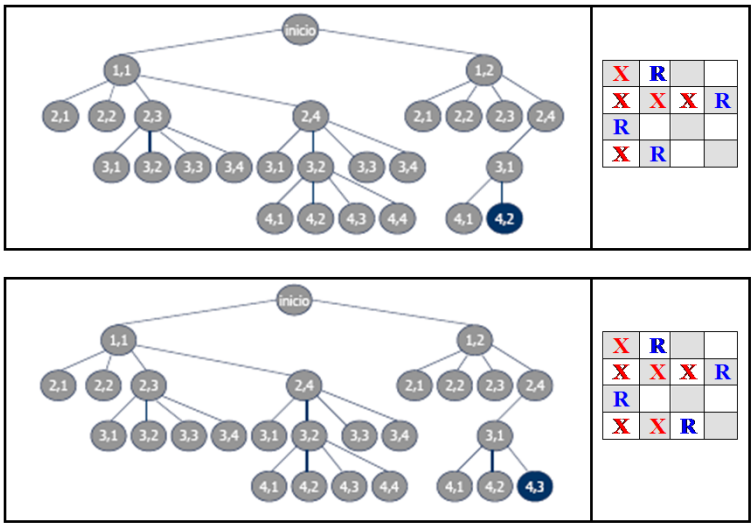
4.5 – Example: Other examples

The n queens problem. Example with 4. State generation.



4.5 – Example: Other examples

The n queens problem. Example with 4. State generation.



Solution. 27 nodes have been tested, without pruning (criteria) 155