DE GRUYTER OPEN Acta Univ. Sapientiae, Informatica **9,** 2 (2017) 144–161

DOI: 10.1515/ausi-2017-0010

Object-oriented backtracking

Tibor GREGORICS
ELTE, Eötvös Loránd University
Budapest
email: gt@inf.elte.hu

Abstract. Several versions of the backtracking are known. In this paper, those versions are in focus which solve the problems whose problem space can be described with a special directed tree. The traversal strategies of this tree will be analyzed and they will be implemented in object-oriented style. In this way, the traversal is made by an enumerator object which iterates over all the paths (partial solutions) of the tree. Two different "backtracking enumerators" are going to be presented and the backtracking algorithm will be a linear search over one of these enumerators. Since these algorithms consist of independent objects (the enumerator, the linear search and the task which must be solved), it is very easy to exchange one component in order to solve another problem. Even the linear search could be substituted with another algorithm pattern, for example, with a counting or a maximum selection if the task had to be solved with a backtracking counting or a backtracking maximum selection.

1 Introduction

The backtracking can solve the tasks that can be made possible to be viewed as a path-finding problem. In this case, the solution of a task is searched in a directed graph which may contain even infinite nodes but the number of the outgoing arcs of each node (i.e. branching factor) is finite. This is the so called

Computing Classification System 1998: F.3.1
Mathematics Subject Classification 2010: 68N30
Key words and phrases: path-finding problem, backtracking algorithm, enumerator, algorithm pattern

δ-graph [10]. The problem space of the task consists of the paths going out from the start node. Among these paths, the solution path must be found, which drives from the start node to any goal node.

There exist several versions of the backtracking algorithm [9] and always that one must be applied which best fits the special features of the directed graph describing the problem [7].

The most general version [5, 7, 10] can be used in arbitrary δ-graphs but it needs a so-called depth bound to terminate for sure. In this way, if there exists a solution path whose length is not greater than the depth bound, the search must find such a solution. However, the determination of the appropriate depth bound is not simple. If this depth bound is too small, a solution will not be found; if it is too big, the computational complexity will grow. The backtracking algorithms over only finite and acyclic directed graphs [5, 7, 10] do not need the depth bound to terminate and they can find solution if there is one.

The best-known versions work in a special directed tree (finite, rooted by the start node, with the same branching factor of the nodes being at the same level). Its branches make up the problem space of the task which is represented by this tree.

This paper focuses on these latest versions of the backtracking. In section 2, those tasks will be defined which can be solved with these versions and there it will be shown how the problem space of these tasks can be symbolized with a special so-called backtracking tree. section 4 presents two different methods that can traverse this backtracking tree in order to enumerate its branches (that is, the elements of the problem space). These traversals are based on well-known concepts but the separation of these traversals and the search will be novel. section 4 shows how these traversals can be implemented as an isolated object-oriented enumerator. In the section 5 a model will be sketched where the backtracking is a collaboration between three objects: a backtracking enumerator, a commonly linear search [3, 4], and the task that is wanted to solve. The advantages of this approach are discussed in section 6.

2 Model of backtracking tasks

The state-space representation is a well-known general modeling technique which can treat the tasks as a path-finding problem [5]. It requires a state space (the set of tuples of values of the essential data corresponding to the task), the initial and final states, and the operators over the state space. The solution of a task is the sequence of operators which can transform the initial

state to any final state. It is obvious that a state space representation can be mapped to a directed graph where the nodes represent the states, and the directed arcs symbolize the effects of the operators. In this way, in order to solve the problem it is enough to find a path which derives from the start node corresponding to the initial state to any goal node describing the final states. This is the solution path.

When a task has got particular features, the following modeling technique, which are a special state space representation, might be used [1, 6, 8, 9].

Definition 1 (Backtracking task) *There are given* n *finite sets (*$n \in \mathbb{N}$*):* $D_1, \ldots, D_n$. *Let us consider the Cartesian product* $D = D_1 \times \ldots \times D_n$. *The aim is to find the element of the set* D *that satisfies the statement* $\rho : D \to \mathbb{L}$ *(*$\mathbb{L} = \{\mathtt{false}, \mathtt{true}\}$*) where this statement can be defined by a series of statetments* $\rho_0, \rho_1, \ldots, \rho_n : D \to \mathbb{L}$ *with the conditions below:*

1. $\rho_0 \equiv \mathtt{true}$
2. $\rho_n \equiv \rho$
3. $\rho_0, \rho_1, \ldots, \rho_n$ *is monotone, i.e.* $\forall u \in D : \rho_i(u) \Rightarrow \rho_{i-1}(u) (i = 1, \ldots, n)$
4. $\rho_i(u)$ *depends on only the first* $i \in \{1, \ldots, n\}$ *components of* $u \in D$*, i.e.* $\forall i \in \{1, \ldots, n\}$ *and* $\forall u, v \in D : \rho_i(u) = \rho_i(v)$ *if* $\forall j, 1 \leq j \leq i : u_i = v_i$.

The tasks that can be modeled in this way are called backtracking tasks. Other path-finding problems might also be solved with backtracking but the backtracking tasks can be solved with a special version of the backtracking algorithm that will be defined below.

Many times, the series $\rho_0, \rho_1, \ldots, \rho_n$ can be defined so that $\forall i \in \{1, \ldots, n\} : \rho_i \equiv \rho_{i-1} \wedge \beta_i$ where $\beta : D \to \mathbb{L}$ and $\beta_i(u)$ depends on the first $i \in \{1, \ldots, n\}$ components of $u \in D$. In this way, the (3) and (4) conditions of the backtracking tasks automatically hold.

A classic example for this model is given by the n-queen problem where n queens must be placed onto an $n \times n$ chessboard without being able to attack each other. One queen attacks any piece in the same row, column and diagonals. Each row of the board must contain exactly one queen. The possible positions of the i^{th} queen put on the i^{th} row are included by the set $D_i = \{0, \ldots, n-1\}$. (As you can see, the columns are numbered from zero up to $n-1$. The reason for this will be clarified soon.) Let us fix that $\rho_0 \equiv \rho_1 \equiv \mathtt{true}$ and $\forall i \in \{2, \ldots, n\}$ and $\forall u \in D : \rho_i(u) = \rho_{i-1}(u) \wedge \forall j \in \{1, \ldots, i-1\} : (u_i \neq u_j \wedge |u_i - u_j| \neq |i - j|)$.

Definition 2 (Backtracking tree) *Let us consider a backtracking task. Its backtracking tree can be constructed in the following way. The nodes on the first level (children of the root) represent the elements of* D_1*. The nodes on the second level symbolize the elements of* $D_1 \times D_2$ *so that the first component of a node on this level is equal to its parent node. In general, the nodes on the* i^{th} *level (*$i = 1, \ldots, n$*) are the elements of* $D_1 \times \ldots \times D_i$*. The branching factor of the nodes on the* $i-1^{\text{th}}$ *level is the cardinality of the set* D_i*. The first* $i-1$ *components of a node on the* i^{th} *level give just the parent of this node; in other words, the parent node is the prefix of its children. The leaves of this tree are the elements of set* D*.*

According to this definition, a backtracking task can be treated as a path-finding problem in its backtracking tree. In order to solve this problem, it is sometimes enough to find the leaf which satisfies the statement ρ; sometimes that path (branch) which drives from the root (this is the start node) to the leaf satisfying the statement ρ (this is the goal node) must be sought.

Let m_i denote the cardinality of D_i for all $i \in \{1, \ldots, n\}$. Since the elements of D_i might be numbered from 0 up to m_{i-1}, these elements can be referred with their ordinal numbers.

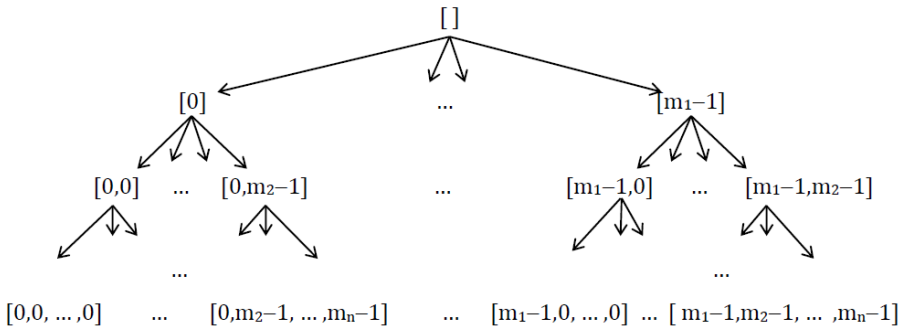

Figure 1: Backtracking tree

This serialization of the set D_i defines a bijection between the set D and the set $\{0, \ldots, m_1 \cdot m_2 \cdots m_n - 1\}$. One element of D can be mapped to a number in a positional numeral system in mixed bases. The base of the i^{th} digit of such numbers is $m_{i+1} \cdots m_n$ and the value of the i^{th} digit might be between 0 and $m_i - 1$ for all $i \in \{1, \ldots, n\}$. It is obvious that the value of an n-digit

natural number is between 0 and $m_1 \cdot m_2 \cdots m_n - 1$. Formally, if $\nu_i : D_i \to \{0, \ldots, m_{i-1}\}$ is a bijection ($i \in \{1, \ldots, n\}$), then $\nu : D \to \{0, \ldots, \prod_{i=1\ldots n} m_i\}$ where $\forall u \in D : \nu(u) = \sum_{i=1\ldots n} \nu_i(u_i) \cdot \prod_{j=i+1\ldots n} m_j$ is a bijection, too. Finally, let us denote ϕ the inverse of ν.

Thus each node on the i^{th} ($i \in \{1, \ldots, n\}$) level of the backtracking tree can be substituted with a code which is a natural number in a positional numeral system in mixed bases. The base of the j^{th} digit of this number is $m_{j+1} \cdots m_i$ and the value of the j^{th} digit might be between 0 and m_{j-1} for all $j \in \{1, \ldots, i\}$. The root node is labeled with the special blank.

This backtracking tree can be shown in the Figure 1.

3 Traversals of the problem space

3.1 Depth-first traversal

Perhaps the best known traversals method of the problem space of the backtracking tasks is the depth-first strategy. This strategy could not work only in special directed trees but also in general directed graphs. Nevertheless, we must underline that there is some difference between the well-known depth-first graph-traversal strategy [2] and the depth-first traversal of a backtracking[10]. Firstly, the latter one does not enumerate only the nodes of the graph but it searches for the first appropriate path driving from the start node. Secondly, the standard depth-first graph-traversal explicitly needs the whole graph which must be traversed but a backtracking algorithm uses much less memory: it stores only the current path. This property is very useful when a typical artificial intelligence problem must be solved and the graph of the problem space is so huge (sometimes infinite) that the total graph could not be stored explicitly. Thirdly, the backtracking could not record all nodes of the graph which have been touched earlier. Thus, a node which have been checked might be checked again when the algorithm rediscovers the very node via another path outgoing from the start node. Fortunately, this unpleasant phenomenon cannot occur if the graph is a tree as it can be seen in our case. In the following, the depth-first traversal means this memory-efficient version of the depth-first strategy.

The depth-first traversal of a directed tree means that the search systematically examines the paths outgoing from the root (these are the branches) from left to right. At each moment, only one partial branch (a path) is stored. This is the current path and its last node is the current node. In each step, the traversal tries to go forward (downward in the tree). If it is not able to or it is not worth going forward, it steps back (upward) to the parent node and selects

the next branch outgoing from this parent. If there are no other unchecked branches going from this parent, then it steps backward (upward) while it finds a parent with untested outgoing branches. In order to implement this process, the untested branches outgoing from the nodes of the current path must be recorded.

This depth-first traversal is even easier over the backtracking tree where the nodes are represented with numbers in a numeral system with mixed bases. For this traversal, it is enough to store only the current node that can be represented with an n-length array v ($\mathsf{v} : \mathbb{N}^{\mathsf{n}}$) and the natural number $\mathtt{ind}$. The label of the current node is the $(\mathtt{ind}-1)$-length prefix of v (i.e. $\mathsf{v}[1,\ldots,\mathtt{ind}-1]$) and it is always supposed that the $\rho_{\mathtt{ind}-1}(\phi(\mathsf{v}))$ holds and each element of v after the position $\mathtt{ind}$ is zero. From this information, all outgoing untested branches of the current path can be read out.

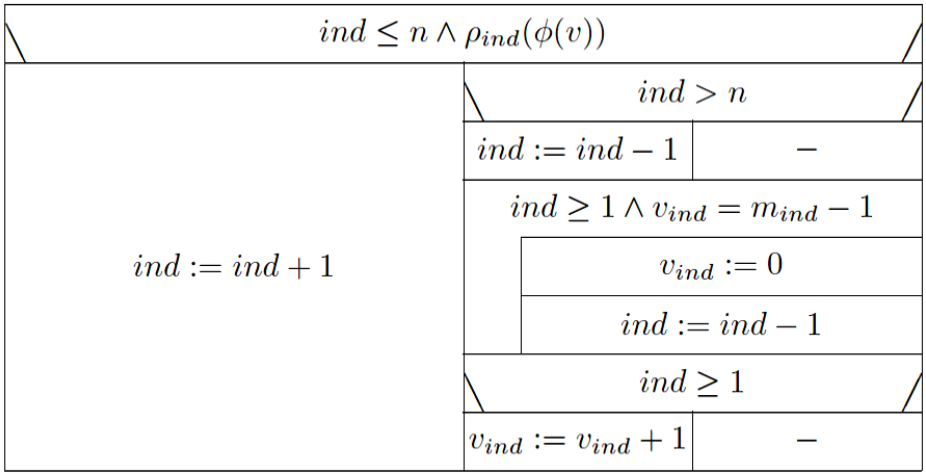

Figure 2: One step of the depth-first traversal

Initially the value of the number $\mathtt{ind}$ is 1 and each element of the array v is zero. The next step of the traversal depends on the statement $\rho_{\mathtt{ind}}(\phi(\mathsf{v}))$ (Figure 2). If it holds and $\mathtt{ind} \leq \mathtt{n}$, then the traversal steps forward with the increase of ind. Otherwise, the traversal steps back – as if the part of the current branch below the $\mathtt{ind}-1^{\text{th}}$ level had been cut – and it looks for the node on the current path that has got an unchecked successor and then selects the first such one as a new current node. The same happens when $\mathtt{ind} = \mathtt{n}+1$,

although, in this case, the traversal does not need to continue since $\rho_n(\phi(\nu))$ holds, i.e. $\rho(\phi(\nu))$ holds. The traversal must stop definitely if $\mathtt{ind} = 0$. This event indicates that the traversal is over.

3.2 Increasing traversal

The problem space of the backtracking tasks in our focus can be traversed with another strategy. It is enough to enumerate the leaves of the backtracking tree (see Figure 1) from left to right. Since the nodes can be represented with natural numbers in a positional numeral system in mixed bases, each leaf is a natural number and they can be generated in increasing order. However, it is not worth enumerating them all one by one because those numbers which may not be goal nodes can be skipped. Yet, how can it be decided without their examination?

Let us suppose that the number ν has been examined and the statement $\rho_{\mathtt{ind}-1}(\phi(\nu))$ holds but the statement $\rho_{\mathtt{ind}}(\phi(\nu))$ does not. It is obvious that each natural number whose first $\mathtt{ind}$ digits are identical to the first $\mathtt{ind}$ digits of ν does not satisfy $\rho_{\mathtt{ind}}$, too. Thus, the enumeration might ignore these numbers. In order to get the next number of the enumeration, it is enough to increase the $\mathtt{ind}^{\text{th}}$ digit of ν by one. In the case when $\rho(\phi(\nu))$ holds there is no index $\mathtt{ind}$ where array ν can be changed, i.e. $\mathtt{ind} = n+1$). If the enumeration should be continued, the value of the variable $\mathtt{ind}$ must be changed to n.

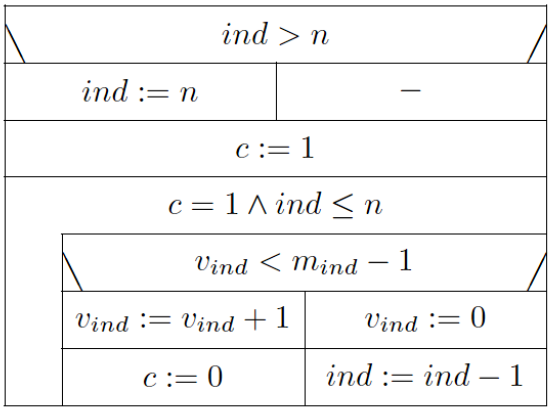

Figure 3: One step of the increasing traversal

This increasing is the task of the algorithm of the Figure 3 [6, 9]. This is a positional addition where variable $\mathtt{ind}$ indicates the position which must be increased. The variable $\mathtt{c}$ contains the carry digit of the addition. If this algorithm terminates with $\mathtt{c} = 0$, then the variable ν will contain the next element of the enumeration. The termination with $\mathtt{c} = 1$ indicates the overflow of the addition and it means that the enumeration is over.

This algorithm must be embedded into the environment which analyzes the current ν and looks for the first $\mathtt{ind}$ for which the statement $\rho_{\mathtt{ind}}(\phi(\nu))$ is false. (If there is no such $\mathtt{ind}$, then $\rho_n(\phi(\nu))$ and thus $\rho(\phi(\nu))$ is true.) This $\mathtt{ind}$ must be passed to the above algorithm.

3.3 Comparison of the traversals

It is worth comparing the two traversal techniques. Let us look at, for example, the problem space of the 4-queens problem (see Figure 4). In this tree, the depth-first traversal moves vertically while the increasing traversal enumerates a part of the leaves horizontally.

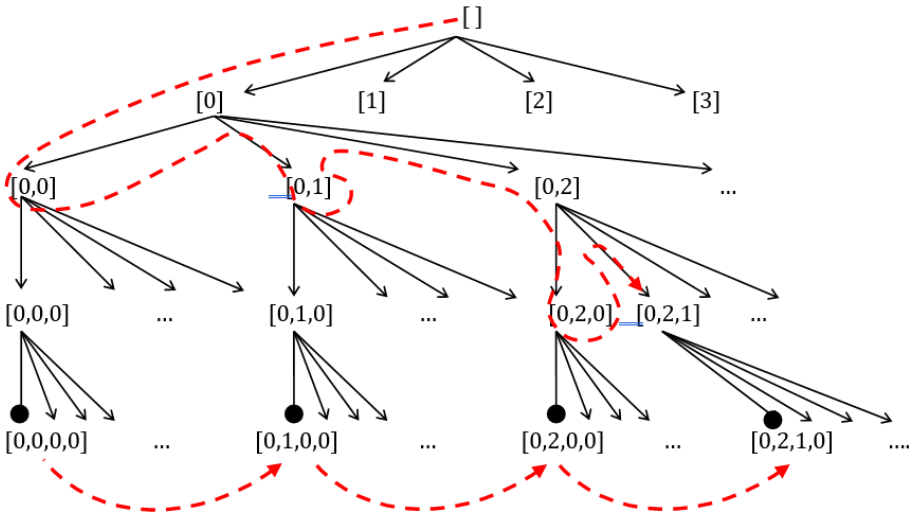

Figure 4: Two kinds of traversals over the 4-queen problem

The backtracking steps can be observed well in the depth-first traversal. If a number of a node at the level $\mathtt{ind}$ does not satisfy the statement $\rho_{\mathtt{ind}}$, i.e. $\rho_{\mathtt{ind}}(\phi(\nu))$ is false where ν is the label of the node, then the traversal

steps back. During the increasing traversal, the backtracking steps are totally hidden. When this enumeration selects a new leaf, then it might jump over several leaves. Each jump is corresponded to the series of backtracking steps of the depth-first traversal which selects the next branch for examination.

The depth-first traversal does not need the examination of ρ. It relies on the statements $\rho_{\mathtt{ind}}$ only. The algorithm of the increasing traversal does not use directly the statements $\rho_{\mathtt{ind}}$ but it is supposed that, before calling this algorithm, the $\rho(\phi(v))$ has been examined. If it is false, the statement $\rho_{\mathtt{ind}-1}(\phi(v))$ holds but the statement $\rho_{\mathtt{ind}}(\phi(v))$ does not; this $\mathtt{ind}$ must be given to the algorithm as input parameter beside the v.

4 Backtracking enumerators

Now the above traversals are going to be implemented as independent enumerator objects. These enumerators iterate over the elements of the problem space.

The problem space ($D = D_1 \times \ldots \times D_n$) can be modeled by the class `Task` (see Figure 5). This class provides the method `rho()` which can decide whether an element satisfies ρ_i or does not. Certainly, this method is abstract; it must be overridden when the concrete task becomes known. A task can be represented by the pair of array v and array m. These are the members of the class `Task`. The array v contains one element of the set of D. The $m[i]$ gives upper limit of the elements of $v[i]$, i.e. for all $i \in [1, \ldots, n] : 0 \leq v[i] < m[i]$. This is the invariant of this representation.

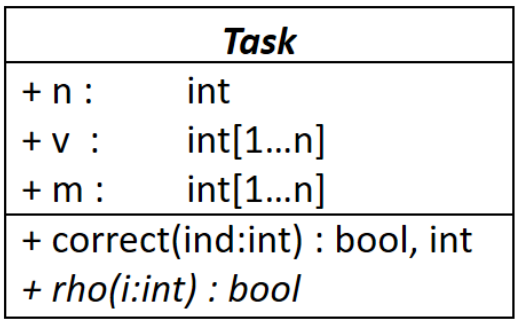

Figure 5: Abstract class of the backtracking task

This class is extended with the method `correct()` that decides whether the

current element of the problem space satisfies the statement $\rho(\phi(u.v))$ or does not.

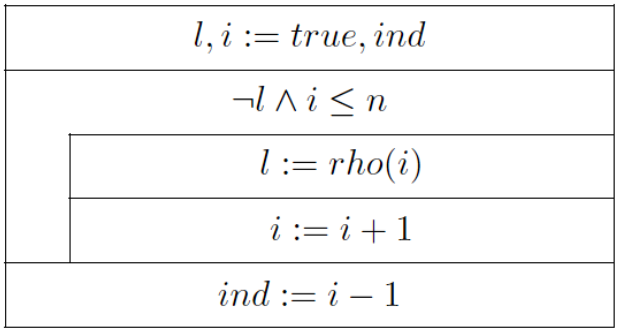

Figure 6: The method correct()

This examination is a special (optimistic) linear search [3, 4] which must check $\rho(i)$ where i goes from 0 up to n and must give the first i for which $\rho(i)$ is false. This process can be accelerated if the index ind ($ind \in [1,\ldots,n]$) is known where $\rho_{ind-1}(\phi(u.v))$ is true. In this case, it is enough to start the search from the index ind instead of 1 (see Figure 6). If $\rho(\phi(u.v))$ is false, it is useful to give back the ind where $rho(ind-1)$ is true but $rho(ind)$ is false.

4.1 Depth-first enumerator

The class DepthFirstEnum describes the object of dept-first enumerator (see Figure 7). It provides the enumeration operators: first(), next(), current(), end() [3, 4]. These operators iterate over the partial branches of the backtracking tree of the problem space of the backtracking task.

Each partial branch can be represented with its ending node, which is an element of $D_1 \times \ldots \times D_{ind-1}$. It can be described with the members of the variable u of Task and the variable ind of (N). The values of ind are between 0 and n. Thus these are members of the class DepthFirstEnum. We suppose that $\rho_{ind-1}(\phi(u.v))$ and for all $i \in [ind+1..n] : u.v[i] = 0$. The method end() indicates the end of the traversal. This is implemented by $ind = 0$ when the traversal has stepped back from the root because it could not find a solution. The method current() returns the current node that is represented by the members. The method first() sets the initial values of the members. In the case $n < 1$ the traversal must be finished immediately, i.e. $ind := 0$. Otherwise

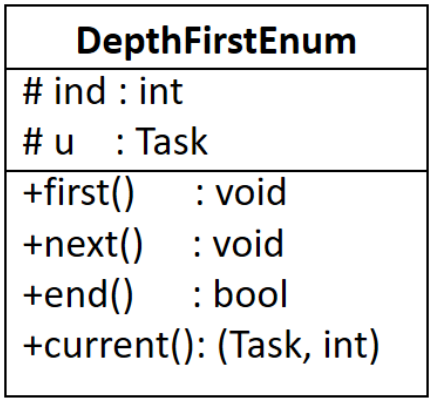

Figure 7: The class of depth-first enumerator

the initialization $u.v := [0, \ldots, 0] : ind := 1$ is needed. The method $next()$ does one step in the problem space according to the depth-first traversal (see Figure 2 with the following corresponding: $u.rho(ind)$ instead of $\rho_{ind}(\phi(v))$, $u.v[ind]$ instead of v_{ind}, and $u.m[ind]$ instead of m_{ind}).

4.2 Increasing enumerator

The class $IncreasingEnum$ describes the object of increasing enumerator (Figure 8). It provides the enumeration operators: $first()$, $next()$, $current()$, $end()$. These operators iterate over some leaves of the backtracking tree. These leaves must be enumerated in increasing order according to their value in the positional numeral system in mixed bases, which has been mentioned before.

Each leaf can be represented by the variable u of $Task$. It is worth introducing the member c of $\{0, 1\}$ that is the overflow digit of the increasing process (see Figure 8). Its value 1 indicates the end of the traversal.

The method $end()$ checks the value of overflow digit c. The method $current()$ returns the current leaf. The method $first()$ initializes $u.v$ and c. The enumeration starts with the element described by the number $[0, \ldots, 0]$ (this is the first value of the variable u) and the overflow c is 0 except for the case $n < 1$ when the traversal must be finished immediately, i.e. $c := 1$. The method $next()$ does one step in the problem space according to the increasing traversal (see Figure 2). Its input parameter is the position ind ($ind \in [1, \ldots, n]$), which shows which position of the number u must be increased according to

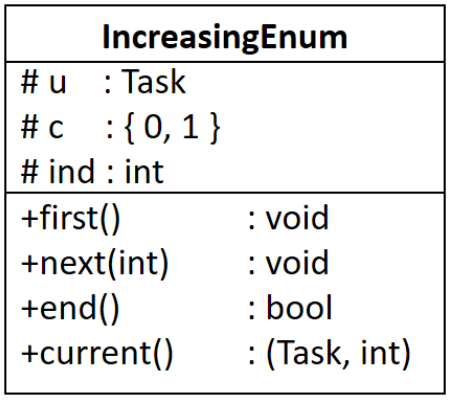

Figure 8: The class of increasing enumerator

the rules of the positional numeral system in mixed bases.

The input of the method `next`() is provided by the method `correct`() (see the class `Task`) and that method requires the index `ind` produced by the method `next`() where $\rho_{ind-1}(\phi(u.v))$ is true but $\rho_{ind}(\phi(u.v))$ is false. Thus the members of the class of the increasing enumerator might be completed with the index `ind` so that either `ind` is zero or $\rho_{ind-1}(\phi(u.v))$ holds. In addition each element of $u.v$ behind the position `ind` is zero. Initially the method `first`() gives the index `ind` a value ($ind := 0$), this index is changed based on its input parameter and then its value is recomputed by the method `next`(), and its value can be queried by the method `current`().

5 Component-oriented backtracking

Based on a backtracking enumerator, the backtracking algorithm can be composed easily. In object-oriented environment, the backtracking algorithm is the result of the cooperation of three objects (see Figure 9): the object of the backtracking enumerator, the object of the task, and the object of the linear search over enumerator [3, 4].

The classes of two kinds of enumerators have been derived from the abstract class `Enumerator` (see Figure 11). This super class includes an object of the class `Task` and an index. The role of this index has been discussed earlier. This index is needed for the method `cond`() of the linear searching.

Under increasing enumeration the method `next`() uses this index: its initial

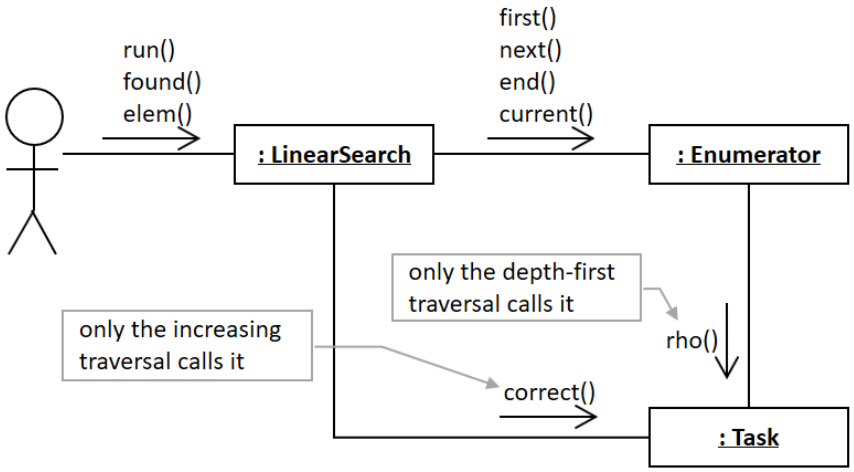

Figure 9: Collaboration of components of the backtracking algorithm

value is got from the linear searching and then the method `next`() modifies it. However, this would be an irregular implementation of the method `next`() because it usually has no external input. Thus – instead of changing the interface of the method `next`() – a "setter" method of the index has been introduced (`setInd`()). We do the same with the extra output of the method `current`(). Since this output is required by the linear search a "getter" method has been also implemented with this very index (`getInd`()). These new methods are defined in the super class `Enumerator`.

The super class `LinearSearch` (see in Figure 11) provides the method `run`() that encapsulates the schema of the algorithm of linear searching, furthermore the method `found`() and the method `elem`() that give the result of the search. Two versions of this algorithm can be differed depending on the way of the traversal. (see Figure 10) Under depth-first traversal the solution can be found if the index of the enumerator is greater than `n`. This makes calling the method `correct`() unnecessary. The index `ind` can be asked from the class `DepthFirstEnum` with its "getter" and the value `n` can be reached through the object `u`. Under the increasing traversal the method `correct`() requires the index of the enumerator (this can be asked with the "getter" of the class `IncreasingEnum`), modifies this index, and gives it to the method `next`() through the "setter" of the enumerator. These differences can be written in

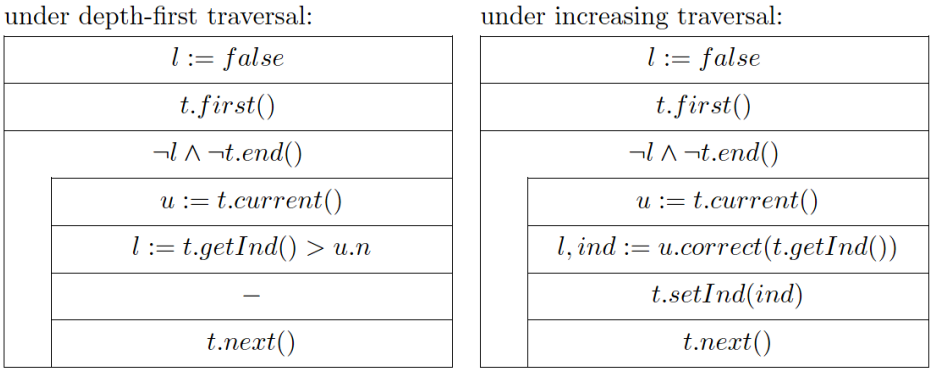

Figure 10: Two versions of linear search

the overridden method `cond()` of the class `DepthFirstLinSearch` and the class `IncreasingLinSearch` thus the method `run()` which calls this method `cond()` can be implemented independently on the way of enumeration. Here the "template function" design pattern is applied. (We must remark that the assignment `u := t.current()` in the method `run()` requires a deep copy.)

The backtracking algorithm is an instance of the class `BacktrackSearch` (see Figure 11). It owns the enumerator which includes the task and creates the appropriate object of the linear searching depending on the kind of the enumerator. The method `run()` calls the same named method of linear searching. Here the "bridge" design pattern is applied.

6 Discussion

The fact that the backtracking consists of three, well-separated components makes the algorithm very flexible. By changing components, it is very easy to change the properties of the search.

In order to solve a new task, it is enough to derive a new class from the super class `Task` and only the abstract method `rho(i)` must be overridden. The object `u` of the enumerator will be an instance of this new class while the other two objects (the enumerator and the linear search) do not change; they are reused.

The object of linear search must be exchanged when the task does not look for one solution but it wants to count how many solutions there are or it wants to look for the best solution according to a given point of view. In these cases,

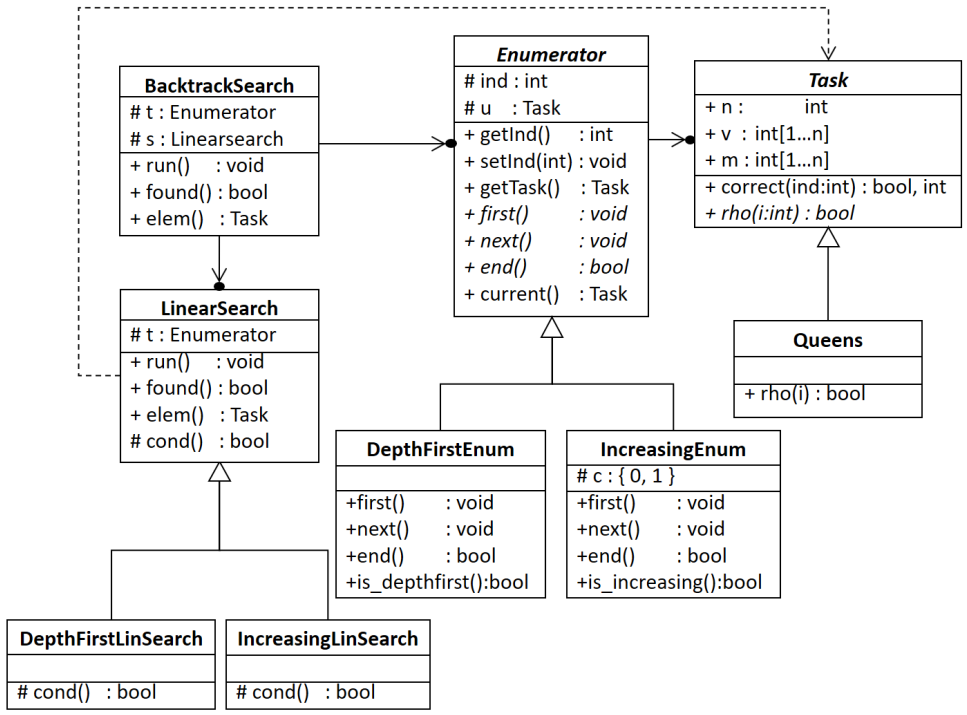

Figure 11: Class diagram of the backtracking algorithm

it is enough to use a counting or a conditional maximum search instead of the linear search.

The "backtracking counting" comes from the counting [3, 4] over enumerators if it uses a backtracking enumerator. Sometimes, only certain solutions must be counted. To solve this task, a logical function $\beta_i : D \to \mathbb{L}$ is needed to check this certain property. (see Figure 12)

The "backtracking maximum search" is the conditional maximum search [3, 4] with a special enumerator. (see Figure 13) The function $f : D \to H$ maps to the well-ordered set H, thus the states of D can be compared.

Only the class `Task` has to be modified if the model of the task which is wanted to solve slightly differs from the model of backtracking tasks. Many times, the problem space of a path-finding task can be described with the directed tree where the number of the outgoing arcs of the nodes can differ on the same level and the goal nodes may be inner nodes.

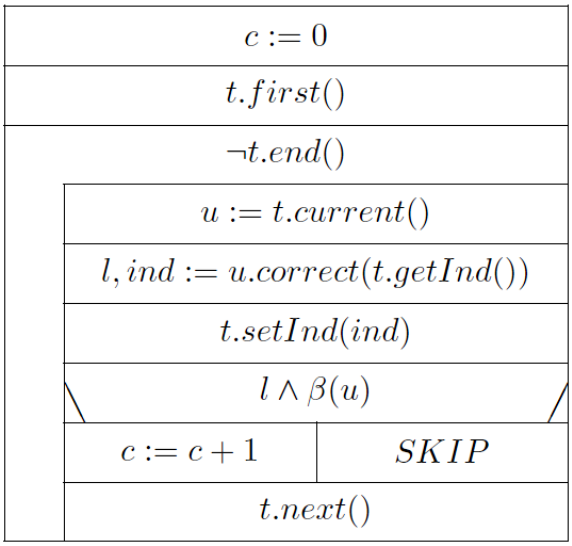

Figure 12: Backtracing counting with increasing enumerator

In the first case, the tree can be extended with alibi nodes so that the branching factor becomes constant inside one level of the tree. Certainly, the semantic of the statement ρ_i must be changed so that it gives false on these alibi (fake) children node.

In the second case, the criterion $\rho \equiv \rho_n$ is substituted with the criterion $\rho \equiv \exists i \in 1, \ldots, n : \rho_i$ or, in a more general way, the criterion $\rho \equiv \exists i \in 1, ldots, n : \rho_i \wedge \gamma_i$ where $\gamma_i : D \to \mathbb{L}$ is the logical function so that the value of $\gamma_i(u)$ depends only on the first i components of the state u. During depth-first traversal, the method `correct()` of the linear search must be overridden as $\gamma_{ind}(u)$. The value of $\gamma_{ind}(u)$ can be computed by an appropriate new method `gamma()` of the class `Task`. During increasing traversal, the method `correct()` must be overridden so that it results in true if there exists an $ind \in 1, \ldots, n$ where $\rho_{ind}(u) \wedge \gamma_{ind}(u)$ holds, otherwise it results in the first index $ind \in 1, \ldots, n$ where $\rho_{ind}(u)$ false.

At the end, we mention the didactical advantage which appears when this component-oriented backtracking is introduced into education. At this stage of the syllabus, the linear search (and other algorithm patterns) has been well known. Only the backtracking enumerator is the novelty. It can help if students have already met the concept of the enumerator; moreover, they have

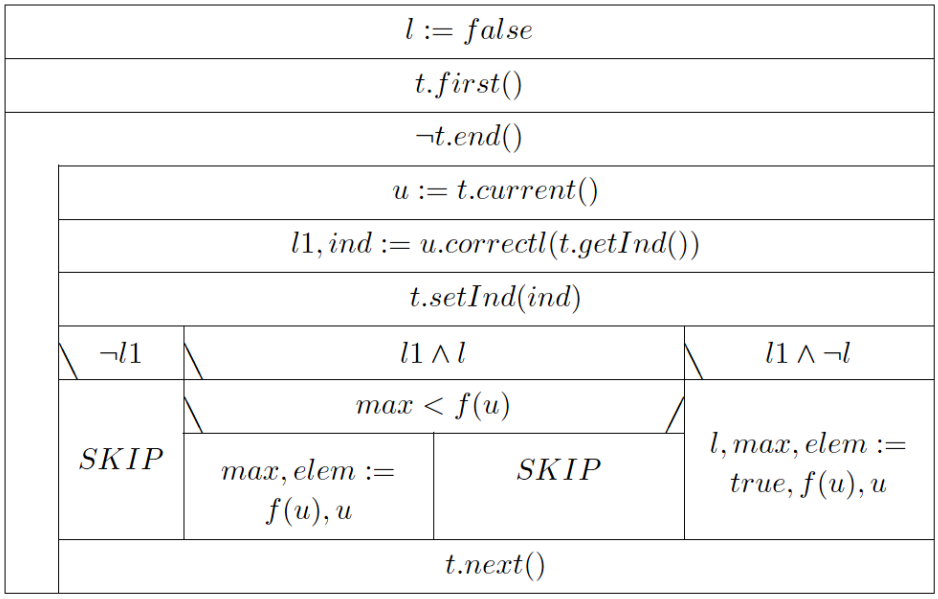

Figure 13: Backtracking maximum search with increasing enumerator

used various enumerators. Certainly, the description of the backtracking tasks which can be solved in this way must be introduced but it is not avoidable in other syllabi.

References

[1] K.A. Berman, J. L. Paul, *Fundamentals of Sequential Algorithms*, PWS Publishing, 1996. ⇒146

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (3rd edition), The MIT Press, 2009. ⇒148

[3] T. Gregorics, *Programozás 1.kötet Tervezés*, ELTE Eötvös Kiadó, 2013. (in Hungarian) ⇒145, 153, 155, 158

[4] T. Gregorics, Programming theorems on enumerator, *Teaching Mathematics and Computer Science*, **8,** 1 (2010) 89–108. ⇒145, 153, 155, 158

[5] I. Fekete, T. Gregorics, S. Nagy, *Bevezetés a mesterséges intelligenciába*, LSI, 1990. (in Hungarian) ⇒145

[6] Á. Fóthi, *Bevezetés a programozásba*, ELTE Eötvös Kiadó, 2005. (in Hungarian) ⇒146, 151

[7] I. Futó (ed), *Mesterséges intelligencia*, Aula, 1999. (in Hungarian) ⇒145
[8] D. E. Knuth, Estimating the efficiency of backtrack programs, *Mathematics of Computation* **29** (1975) 121–136. ⇒146
[9] K. I. Lörentey, Fekete, Á. Fóthi, T. Gregorics, On the wide variety of backtracking algorithms, *ICAI'05 Eger, Hungary, January 28–February 3.* 2005, pp. 165–174. ⇒145, 146, 151
[10] N. J. Nilsson, *Principles of Artificial Intelligence*, Springer-Verlag, Berlin, 1982. ⇒145, 148

Received: November 2, 2017 • Revised: November 27, 2017