Algorithm Design Patterns: Programming Theory Perspective

(Extended Abstract)

Nikolay V. Shilov

A.P. Ershov Institute of Informatics Systems, Russian Academy of Sciences
Lavren'ev av. 6, 630090 Novosibirsk, Russia
shilov@iis.nsk.su
http://persons.iis.nsk.su/en/person/shilov

Abstract. Design and Analysis of Computer Algorithms is a must of Computer Curricula. In particular it teaches algorithm design patterns like *greedy method*, *divide-and-conquer*, *dynamic programming*, *backtracking* and *branch-and-bound*. Unfortunately, all listed design patterns are taught, learned and comprehended by examples, while they can be formalized as design templates, rigorously specified, and mathematically verified. Greedy method is the only pattern that had been studied from rigour mathematical point of view in XX century. Later, the author published (in years 2010-2012 in separate papers) formalization, specification and verification for three more patterns, namely *dynamic programming*, *backtracking* and *branch-and-bound*. In the present extended abstract these studies are summarized and discussed from programming theory perspective using concepts and techniques used in *Abstract Data Types*, *Theory of Program Schemata*, *Partial and Total Correctness*, *program specialization*.

To commemorate 85 anniversary of A.P. Ershov (1931-1988), a Scientist that drew my interest to Theory of Programming.

1 Introduction

Algorithm design patterns (ADP) like *greedy method*, *divide-and-conquer*, *dynamic programming* (DYN), *backtracking* (BTR) and *branch-and-bound* (B&B) are usually considered as Classics of the Past (going back to days of R. Floyd and E. Dijkstra). However, ADP can be (semi)formalized as design templates, specified by correctness conditions, and formally verified either in the Floyd-Hoare methodology, by means of the Manna-Pnueli proof-principles, or in some other way.

Nevertheless until 2010 the only formalized method of algorithm design was greedy method (or greedy algorithms): it was proven in years 1971-1993 [5,13,15] that if the structure of the domain in an optimization problem is a *matroid* (or, is more general, *greedoid*), then application of greedy algorithm guarantees a global optimum for the problem.

Unfortunately, further progress with DYN, BTR and B&B techniques has degenerated into an extensive collection of "success stories" and "recipes" how they have been used in the context of particular combinatorial or optimization problems. This leads to educational situation when the most popular contemporary textbooks on the algorithm design and implementation look like Cooking Books [1, 4].

BTR and B&B are widely used in design of combinatorial algorithms for (virtual) graph traversing. In particular, most global optimization methods using interval techniques employ a branch-and-bound strategy [9] These algorithms decompose the search domain into a collection of boxes, arrange them into a tree-structure (according to inclusion), and compute the lower bound on the objective function by an interval technique. Basically the strategy is an algorithm design pattern that originates in graph traversal.

In general graph traversal refers to the problem of visiting all the nodes in a (di)graph to find particular nodes (vertices) that enjoy some property specified by some Boolean "criterion condition". A Depth-first search (DFS) is a technique for traversing a finite graph that visits the child nodes before visiting the sibling nodes. A Breadth-first search (BFS) is another technique for traversing a finite undirected graph that visits the sibling nodes before visiting the child nodes.

Sometimes it is not necessary to traverse all vertices of a graph to collect the set of nodes that meet the criterion function, since there exists some Boolean "boundary condition" which guarantees that child nodes do not meet the criterion function: Backtracking (BTR) is DFS that uses boundary condition, branch-and-bound (B&B) is DFS that uses boundary condition. Backtracking became popular in 1965 due to research of S.W. Golomb and L.D. Baumert [11], but it had been suggested earlier by D. H. Lehmer. Branch-and-bound was suggested by A.H. Land and A.G. Doig in 1960 [18].

Formalization and verification of backtracking and branch-and-bound ADP was attempted in years 2011-2012: a unified ADP for BTR and B&B was formalized as a design template, specified by correctness conditions, and formally verified by means of the Manna-Pnueli proof-principles first [20] and later in the Floyd-Hoare methodology [23].

Dynamic Programming was introduced by Richard Bellman in the 1950s [3] to tackle optimal planning problems. At this time, the noun *programming* had nothing in common with more recent *computer programming* and meant *planning* (compare: *linear programming*). The adjective *dynamic* points out that Dynamic Programming is related to a *change of state* (compare: *dynamic logic*, *dynamic system*). *Bellman equations* are recursive functional equalities for the objective function that express the optimal solution at the current state in terms of optimal solutions at changed states. They formalize the following *Bellman Principle of Optimality*: an optimal program (or plan) remains optimal at every stage.

At the same time, according to [7], R. Bellman, speaking about the 50s, explains:

> *An interesting question is, "Where did the name,* dynamic programming*, come from?" The 1950s were not good years for mathematical research.*

> *(...) Hence, I felt I had to do something to shield [the Secretary of Defense] and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. (...) Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.*

A preliminary formalization of Dynamic Programming has been published in [21] and then in [22]. The approach

- formalizes *descending* Dynamic Programming ADP by a *recursive program scheme* (with variable arity of functional symbols),
- formalizes *ascending* Dynamic Programming as a *standard program scheme* (also with variable arity of functional symbols) augmented by *generic dynamic array* to compute the least fix-point (according to the Knaster-Tarski theorem),
- proves functional equivalence of both schemes, and proves that in general case we can't rid of *dynamic memory* when implement the ascending dynamic programming pattern.

This extended abstract represents

in the next section 2: the unified template for Backtracking and Branch-and-Bound, its (semi-)formal specification and main correctness statements (as in [20, 23]);

in the section 3: the unified templates for descending and ascending Dynamic Programming, their functional equivalence statement (as in [21, 22]).

The final section 4 discusses in brief examples of use of the templates and further research topics (in particular, *dynamic memory* issues and *partial evaluation* perspective).

2 Template for Backtracking and Branch-and-Bound

2.1 Abstract Data Type "Teque"

Let us define a special temporal abstract data type (ADT) "theque[1]" for the unified representation of BTR and B&B. Theque is a finite collection (i. e. a set) of values (of some background data type) marked by disjoint "time-stamps". The time stamps are readings of "global clock" that counts time in numbers of "ticks", they (time-stamps) never change and always are not greater than current reading of the clock. Let us represent an element x with a time-stamp t

[1] Theque – storage (Greek: theke), e. g. "discotheque".

by the pair (x, t). Readings of the clock as well as time-stamps are not "visible" for any "observer". Let us assume that this "tick" is indivisible, every action takes a positive (integer) number of ticks, and the clocks never resets or restarts.

ADT theque inherits some set-theoretic operations: the *empteq* (i. e. "empty teque")) is simply the empty set ($\emptyset$), set-theoretic equality ($=$) and inequality ($\neq$), subsumption ($\subset$, $\subseteq$). At the same time ADT theque has its own specific operations, some of these operations are time-independent, some others — time-sensitive, and some are time-dependent. Let us enumerate below time-independent operations, and describe time-dependent and time-sensitive operations in the next paragraphs.

- Operation Set: for every teque T let $Set(T)$ be $\{x \ : \ \exists t((x,t) \in T)\}$ the set of all values that belongs to T (with any time-stamp).
- Operations In and Ni: for every teque T and any value x of the background type let $In(x,T)$ denote $x \in Set(T)$, and let $Ni(x,T)$ denote $x \notin Set(T)$.
- Operation $Spec$ (*specification*): for every teque T and any predicate $\lambda x.Q(x)$ of values of the background type let teque $Spec(T,Q)$ be the following subteque $\{(x,t) \in T \ : Q(x)\}$.

The unique time-dependent operation is a synchronous addition $AddTo$ of elements to teques. For every finite list of teques T_1, ... T_n ($n \geq 1$) and finite set $\{x_1, \ ... \ x_m\}$ of elements of the background type ($m \geq 0$), let execution of $AddTo(\{x_1, \ldots x_m\}, \ T_1, \ldots T_n)$ at time t (i. e. the current reading of the clock is t) returns n teques T'_1, ... T'_n such, that there exist m moments of time (i. e. readings of the clock) $t = t_1 < ... < t_m = t'$ such that t' is the moment of termination of the operation, and for every $1 \leq i \leq n$ the teque T'_i expands T_i by $\{(x_1,t_1), \ ... \ (x_m,t_m)\}$, i. e. $T'_i = \ T_i \cup \ \{(x_1,t_1), \ ... \ (x_m,t_m)\}$. Let us observe that this operation is non-deterministic due to several reasons: first, the set of added elements $\{x_1, \ ... \ x_m\}$ can be sorted in different manners; next, time-stamps $t_1 < ... < t_m$ can be arbitrary (starting at the current time). Let us write $AddTo(x, T_1, \ldots T_n)$ instead of $AddTo(\{x\}, T_1, \ldots T_n)$ in the case of a singleton set $\{x\}$.

There are three pairs of time-sensitive operations: Fir and $ReMFir$, Las and $RemLas$, Elm and $RemElm$. Let T be a teque. Recall that all values in this teque have disjoint time-stamps.

- Let $Fir(T)$ be the value of the background type (i. e. without a time-stamp) that has the smallest (i. e. the first) time-stamp in T, and let $RemFir(T)$ be the teque that results from T after removal of this element (with the smallest time-stamp).
- Let $Las(T)$ be the value of the background type (i. e. without a time-stamp) that has the largest (i. e. the last) time-stamp in T, and let $RemLas(T)$ be the teque that results from T after removal of this element (with the largest time-stamp).

We also assume that $Elm(T)$ is "some" element of T (also without any time-stamp) that is defined according to some "procedure" (unknown for us) and $RemElm(T)$ is the teque that results from T after removal of this element (with its time-stamp).

2.2 Unified Template

Let us introduce some notation that unifies representation of BTR and B&B by a single template for graph traversing: let FEL and REM stay either for Fir and $ReMFir$, or for Las and $RemLas$, or for Elm and $RemElm$. It means, for example, that if we instantiate Fir for FEL, then we must instantiate Fir for FEL and $RemFir$ for REM throughout the template. Instantiation of Fir and $RemFir$ imposes a queue discipline "first-in, first-out" and specializes the unified template to B&B template; instantiation of Las and $RemLas$ imposes a stack discipline "first-in, last-out" and specializes the template to BTR template; instantiation of Elm and $RemElm$ specializes the unified template to "Deep Backtracking" or "Branch and Bounds with priorities" templates.

Let us say that a (di)graph is concrete, if it is given by the enumeration of all vertices and edges, or by the adjacency matrix, or in any other explicit manner. In contrast, let us say that a (di)graph G is virtual, if the following features are given:

- a type $Node$ for vertices of G, the initial vertex ini (of this type) such that every vertex of G is reachable from ini;
- a computable function $Neighb: \ Node \rightarrow 2^{Node}$ that for any vertex of G returns the set of all its neighbors (children in a digraph).

In this notation a unified template for traversing a virtual graph G with the aid of "easy to cheque"

- a boundary condition $B: \ 2^{Node} \times Node \ \rightarrow BOOLEAN$,
- and a decision condition $D: \ 2^{Node} \times Node \ \rightarrow BOOLEAN$

for collecting all nodes that meet a "hard to cheque"

- criterion condition $C : Node \rightarrow BOOLEAN$

can be represented by the following pseudo-code.

```
VAR U: Node;
VAR S: set of Node;
VAR Visit, Live, Out: teque of Node;
Live, Visit:= AddTo(ini, empteq, empteq);
Out:= empteq; IF D({ini}, ini) THEN Out:= AddTo(ini, Out);
WHILE Live ≠ empteq
      DO U:= FEL(Live); Live:= REM(Live);
        S:= {W ∈ Neighb(U) : Ni(W, Visit) & ¬B(Set(Visit), W)};
        Live, Visit:= AddTo(S, Live, Visit);
        Out:= Spec(Out, λx.D(Set(Visit), x));
        IF D(Set(Visit), U) THEN Out:= AddTo(U,Out);
      OD
```

2.3 Correctness

An algorithm without specification is a tool without manual: no idea how to use it and what to expect. A specified algorithm without correctness proof is a non-certified tool, it can be dangerous in use. So we have to specify and prove correctness of our unified template. We would like to use Floyd – Hoare approach to algorithm specification and proof [2, 10]. In this approach an algorithm is specified by a precondition and a postcondition for input and output data, correctness is proved with the aid of loop invariants by induction.

The postcondition is simple: Teque *Out* consists of all nodes of the graph G (with time-stamps) that meet the criterion condition C, and each of these nodes has a single entry (occurrence) in *Out*.

The precondition is more complicated, and can be presented as a conjunction of the following clauses.

1. G is a virtual (di)graph, *ini* is a node of G, $Neighb$ is a function that computes for every node the set of all its neighbors so, that all nodes of G can be reached from *ini* by iterating $Neighb$.
2. For every node x of G the boundary condition $\lambda S.B(S,x)$ is a monotone function: $B(S_1,x) \Rightarrow B(S_2,x)$ for all sets of nodes $S_1 \subseteq S_2$ (i. e. if a node is ruled-out, then it is ruled-out forever).
3. For all nodes x and y of G, for any set of nodes S, if y is reachable from x, then $B(S,x)$ implies $B(S,y)$ (i. e. if a node is ruled-out then all its successors are ruled out also).
4. For every node x of G, the decision condition $\lambda S.D(S,x)$ is an anti-monotone function: $D(S_2,x) \Rightarrow D(S_1,x)$ for all sets of nodes $S_1 \subseteq S_2$ (i. e. a candidate node may be discarded later).
5. For every set of nodes S, if $S \cup \{x \in G : B(S,x)\}$ is equal to the set of all nodes of G, then $D(S,x) \Leftrightarrow C(x)$ (i. e. the decision condition D applied to a set with "complete extension" is equivalent to the criterion condition C).

Proposition 1. *The unified template is partially correct with respect to the above precondition and postcondition, i. e. if the input data meet the precondition and a particular algorithm instantiated from the template terminates on the input data, then it terminates with the output that meets the postcondition.*

Proposition 2. *If the input graph is finite then the unified template eventually terminates, i. e. every particular algorithm instantiated from the template always halts traversing the graph after a finite number of steps.*

The above two propositions imply the following total correctness statement for Backtracking and Branch-and-Bound.

Corollary 1.
If the input data (including the boundary, decision and criterion conditions B, D and C) meet the precondition, and the virtual graph G for traversing is finite, then every particular algorithm instantiated from the template terminates after $O(|G|)$ iterations of the loop, so that upon termination the set $Set(Out)$ will consist of all nodes of the graph G that meet the criterion condition C.

Remark 1. For proofs please refer papers [20, 23].

3 Templates for Dynamic Programming

3.1 Recursive Descending Dynamic Programming

If to analyse Bellman principle then it is possible to suggest the following *recursive scheme* as a general pattern for Bellman equations:

$$G(x) \;=\; if\ p(x)\ then\ f(x)\ else\ g(x,\ h_i(G(t_i(x)),\ i \in [1..n(x)])), \qquad (1)$$

where

- G is a function variable to represent an objective function from some domain X to some range Y that are to be optimized;
- p is a predicate symbol to represent a known predicate over the same domain X as above;
- f is a function symbol to represent a known function from the same domain X to the same range Y;
- g is a function symbol to represent a known operation with a variable arity on the same domain X (i.e. a function from X^* to X);
- all h_i and t_i, $i \in \mathbb{N}$, are functional symbols to represent known functions from the range Y to the domain X (in case of h_i) and to represent known operations on the domain X (in case of t_i).

Here we understand the recursive scheme in the sense of the *theory of program schemata* [8, 16, 19]. Let us refer the above recursive scheme as a *recursive template for descending Dynamic Programming.*

3.2 Iterative Ascending Dynamic Programming

Let us consider a function $G : X \to Y$ that is defined by the interpreted recursive scheme (1) of Dynamic Programming. For every argument value $v \in X$, such that $p(v)$ doesn't hold, let *base* be the following set $bas(v)$ of values $\{t_i(v) : i \in [1..n(v)]\}$. Let us remark that for every argument value v, if $G(v)$ is defined, $bas(v)$ is finite. Let us also observe that if the objective function G is defined for some argument value v, then it is possible to *pre-compute* (i.e. compute prior to the computation of $G(v)$) the *support* for this argument value v, i.e. the set $spp(v)$ of all argument values that occur in the recursive computation of $G(v)$, according to the following recursive algorithm

$$spp(x) \;=\; if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup (\bigcup_{y \in bas(x)} spp(y)). \qquad (2)$$

Another remark is that for every argument value v, if $G(v)$ is defined, then $spp(v)$ is finite (since computation of $G(v)$ terminates). Let us say that a function $SPP : X \to 2^X$ is an *upper support approximation* if for every argument value v, the following conditions hold:

- $v \in SPP(v)$,

– $spp(u) \subseteq SPP(v)$ for every $u \in SPP(v)$,
– if $spp(v)$ is finite then $SPP(v)$ is finite.

Let us consider the case when some upper approximation is *easier to compute*, i.e. the (time and/or space) complexity of the available algorithm to compute it is better than the complexity of the available algorithm that computes G. Then it makes sense to use *iterative ascending Dynamic Programming*.

Ascending Dynamic Programming comprises the following steps.

1. Input argument value v and compute $SPP(v)$. Let G be an array (in Pascal style) *var* G : Y *array of* $SPP(v)$ of Y-values indexed by values in $SPP(v)$. Then compute and save in the array G values of the objective function G for all arguments $u \in SPP(v)$ such that $p(u)$: $G[u] := f(u)$.
2. Expand the set of saved values of the objective function by values that can be immediately computed on the basis of the set of saved values: for every $u \in SPP(v)$, if $G(u)$ has not been computed yet, but for every $w \in bas(u)$ the value $G(w)$ has already been computed and saved in $G[w]$, then compute and save $G(u)$ in $G[u]$: $G[u] := g(u, (h_i(G(t_i(u))), i \in [1..n(u)]))$.
3. Repeat Step 2 until the moment when the value of the objective function for the argument v is saved.

The ascending Dynamic Programming is an imperative iterative procedure.

3.3 Formalization

Let us formalize iterative ascending Dynamic Programming by means of an imperative pseudo-code annotated by precondition and postcondition [10].

Precondition:
D is a non-empty set of argument values,
S and P are "trivial" and "target" subsets in D,
$F : 2^D \to 2^D$ is a call-by-value total monotone function,
$\rho : 2^D \times 2^D \to Bool$ is a call-by-value total function monotone on the second argument.

Peseudo-code:

```
VAR U= S, V: subsets of D;
REPEAT V:= U; U:= F(V)∪S UNTIL (ρ(P,U) or U=V)
```

Postcondition: $\rho(P,U) \Leftrightarrow \rho(P,T)$,
where U is the final value of the variable `U` upon termination
and T is the least set of D such that $T = (F(T) \cup S)$.

Here the initialization `U=` S corresponds to the first step of the informal description of ascending Dynamic Programming, the second assignment `U:=` F`(V)` in the loop body corresponds to the second step, and the loop condition ρ`(`P`,U)` corresponds to the condition at the third step; an auxiliary variable `V`, the first assignment `V:= U` and condition `U=V` are used for termination in case when no further progress is possible.

We would like to refer to this formalization as the *iterative ascending Dynamic Programming template*.

3.4 Correctness and Equivalence

Proposition 3. (Knaster-Tarski fix-point theorem [14])
Let D be a non-empty set, $G : 2^D \to 2^D$ be a total monotone function, and R_0, R_1, ... be the following sequence of D-subsets: $R_0 = \varnothing$ and $R_{k+1} = G(R_k)$ for every $k \geq 0$. Then there exists the least fix-point $T \subseteq D$ of the function G and $R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots R_k \subseteq R_{k+1} \subseteq \ldots \subseteq T$.

The following two propositions are trivial consequences of the above one.

Proposition 4.
Iterative ascending Dynamic Programming template is partially correct.

Proposition 5. *Assume that for some input data the precondition of the iterative ascending Dynamic Programming template is valid and the domain D is finite. Then the algorithm generated from the template terminates after $|D|$ iterations of the loop.*

In turn they imply the following (functional) equivalence statement for recursive (descending) and iterative (ascending) Dynamic Programming.

Corollary 2.
Let G be an arbitrary function defined by interpreted recursive scheme 1. Let

- *D be a* generic dynamic array $\{(u, G(u)) : u \in SPP(v)\}$*, where $SPP(v)$ is an upper support approximation;*
- *S be $\{(u, f(u)) : p(u)$ and $u \in SPP(v)\}$ and P be a singleton $\{(v, G(v))\}$;*
- *F be $\lambda Q \subseteq D.\ \{(u, w) \in D \mid n = n(u),$*
  $\exists w_1, \ldots w_n : (t_1(u), w_1), \ldots (t_n(u), w_n) \in Q,$
  and $w = g(u, h_1(w_1), \ldots h_n(w_n))\}$;
- *ρ be $\lambda Q.(P \subseteq Q)$ (that is equivalent to $\lambda Q \subseteq D.(\exists w : (v, w) \in Q)$).*

Then, the algorithm resulting from the iterative ascending Dynamic Programming template after the specified specialization is totally correct, the final value U of the variable `U` contains a single pair (v, y), and y in this pair equals to $G(v)$.

Remark 2. For proofs please refer paper [22].

4 Conclusion

We have represented in this extended abstract the unified template for Backtracking and Branch-and-Bound, and templates for recursive (descending) and iterative (ascending) Dynamic Programming, specified these templates by means of (semi-formal) precondition and postcondition, stated the total correctness and functional equivalence (for Dynamic Programming) for the templates. "Manual" formal proofs of propositions and corollaries stated in this extended abstract can be found in papers [20–23].

In the cited papers [20–23] one can find the following examples of specialisation of the presented templates:

- in papers [20, 23] unified BTR and B&B template was specialized to solve Discrete Knapsack Problem,
- in papers [21, 22] Dynamic Programming templates were specialised to solve a toy sample Dropping Bricks Puzzle as well as for solving finite position games and to Cocke-Younger-Kasami algorithm for parsing of context-free language.

Basically, the primary purpose of the *specified* and *verified* templates for algorithm design patterns is to use them for (semi-)automatic *specialization* of the patterns to generate correct but more *efficient* algorithms to solve concrete problems. One may observe that this purpose is closely related to purpose of *Mixed Computations* [6] and/or *Partial Evaluation* [12]; the difference consists in level of consideration: in our case we speak about *algorithm design* and use *pseudo-code* while in Mixed Computations and Partial Evaluation *programming languages* and *program code* are in use. Nevertheless further studies of algorithm design templates from Mixed Computations and Partial Evaluation perspective may be an interesting research topic. In particular, it may be interesting to try to building-in algorithm design templates into an educational IDE (Integrated Development Environment) to support (semi-)automatic algorithm generation.

Another interesting topic for theoretical research is a need of *dynamic memory* either for teque or generic array implementation. In particular, we demonstrated that every function defined by the recursive scheme of Dynamic Programming may be computed by an iterative program with a *generic dynamic array*. The advantage of the translation is the use of an *array* instead of a *stack* generally required to translate recursion. Nevertheless a natural question arises: may *finite static memory* suffices for computing this function? Unfortunately, in general case the answer is negative according to the following proposition proved by M.S. Paterson and C.T. Hewitt [16, 19].

Proposition 6. *The following special case of general recursive scheme of descending Dynamic Programming (1)*

$$F(x) = \ if\ p(x)\ then\ x\ else\ f(F(g(x)), F(h(x)))$$

is not equivalent to any standard program scheme (i.e. an uninterpreted iterative program scheme with finite static memory).

This statement does not mean that dynamic memory is *always* required; it just means that for *some* interpretations of *uniterpreted* symbols p, f, g and h the size of required memory depends on the input data. But if p, f, g and h are *interpreted*, it may happen that function F can be computed by an iterative program with a finite static memory. For example, it is possible to prove that Dropping Bricks Puzzle [22] can be solved by an iterative algorithm with two integer variables. Two other examples of this kind are the factorial function and Fibonacci numbers

$$Fac(n) = \ if\ n = 0\ then\ 1 else\ n \times Fac(n-1),$$

$$Fib(n) = \ if\ n = 0\ or\ n = 1\ then\ 1\ else\ Fib(n-2) + Fib(n-1);$$

they both match the pattern of scheme in the above proposition, but three integer variables suffice to compute them by iterative programs. So, the next problem for further research is about those of functions that can be computed with finite static memory (i.e. like the optimal number of bricks droppings, factorial values or Fibonacci numbers) by iterative imperative algorithms generated from ascending dynamic programming.

Another research topic is about use of different fix-points in the Dynamic Programming context. A unified logical approach to a variety of fix points can be found in [17]; a natural question follows: what if to use for algorithm design other fix-points (studied in the cited paper) than the least one?

References

1. Aho A.V., Hopcroft J. E., Ullman J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. Apt K.R., de Boer F.S., Olderog E.-R. *Verification of Sequential and Concurrent Programs.* Third edition. Springer, 2009.
3. Bellman, R. *The theory of dynamic programming.* Bulletin of the American Mathematical Society, 1954, v.60, p.503-516.
4. Cormen T.H., Leiserson C.E., Rivest R.L., and Stein C. *Introduction to Algorithms.* Third edition. The MIT Press, 2009.
5. Edmonds J. *Matroids and the greedy algorithm.* Mathematical Programming, 1971, v.1, p.127113.
6. Ershov A.P. *Mixed computation: potential applications and problems for study.* Theor. Comp. Sci., 1982, v18(1), p.41-67.
7. Gimbert, H. *Games for Verification and Synthesis.* Slides for 10th School for young researchers about Modelling and Verifying Parallel processes (MOVEP). — `http://movep.lif.univ-mrs.fr/documents/marta-slides1.pdf`.
8. Greibach, S.A. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer, 1975. (Lecture Notes in Computer Science, v.36.)
9. Gray P., Hart W., Painton L., Phillips C., Trahan M., Wagner J. *A Survey of Global Optimization Methods.* Sandia National Laboratories, 1997, available at `http://www.cs.sandia.gov/opt/survey/main.html`.
10. Gries D. *The Science of Programming.* Springer, 1987.
11. Golomb S.W. and Baumert L.D. *Backtrack Programming.* Journal of ACM, 12(4), 1965, p.516-524.
12. Jones J.D., Gomard C.K., and Sestoft P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993, available at `http://www.itu.dk/people/sestoft/pebook/`.
13. Helman P., Moret B.M.E., Shapiro H.D. *An exact characterization of greedy structures.* SIAM Journal on Discrete Mathematics, 1993, v.6(2), p.274283.
14. Knaster, B., Tarski, A. *Un theoreme sur les fonctions d'ensembles.* Ann. Soc. Polon. Math., 1928, n.6, p.133-134.
15. Korte B., Lovasz L. *Mathematical structures underlying greedy algorithms.* Fundamentals of Computation Theory: Proceedings of the 1981 International FCT-Conference, Szeged, Hungaria, August 2428, 1981, Lecture Notes in Computer Science, 1981, v.117, p.205209.

16. Kotov V.E., Sabelfeld V.K. *Theory of Program Schemata.* (Teoria Skhem Programm.) Science (Nauka), 1991. (In Russian.)
17. Lisitsa A. *Temporal Access to the Iteration Sequences: A Unifying Approach to Fixed Point Logics.* Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck , Germany, September 12-14, 2011. IEEE Computer Society, 2011, p.57-63.
18. Land A. H. and Doig A. G. *An automatic method of solving discrete programming problems.* Econometrica, 28(3), 1960, p.497-520.
19. Paterson M.S., Hewitt C.T. *Comperative Schematology.* Proc. of the ACM Conf. on Concurrent Systems and Parallel Computation, 1970, p.119-127.
20. Shilov, N.V. *Algorithm Design Template base on Temporal ADT.* Proceedings of 18th International Symposium on Temporal Representation and Reasoning, 2011. IEEE Computer Society. P.157-162.
21. Shilov N.V. *Inverting Dynamic Programming.* Proceedings of the Third International Valentin Turchin Workshop on Metacomputation, 2012. Publishing House University of Pereslavl. P.216-227.
22. Shilov N.V. *Unifying Dynamic Programming Design Patterns.* Bulletin of the Novosibirsk Computing Center (Series: Computer Science, IIS Special Issue), v.34, 2012, p.135-156.
23. Silov N.V. *Verification of Backtracking and Branch and Bound Design Templates.* Automatic Control and Computer Sciences, 2012, v.46(7), p.402409.