# “The Theory of Everything”: uniform algorithm design patterns

(backtracking, branch & bound, greedy algorithms, divide and conquer, and dynamic programming...)

Nikolay V. Shilov

Simulated Large Hadron Collider CMS particle detector data depicting a Higgs boson produced by colliding protons decaying into hadron jets and electrons

Part 1

# Introduction

# Standard Template Library

- Story (borrowed from Standard Template Library – Wikipedia) that I didn't know till 2021 (and firstly learnt from Eugene Zouev lectures on *Software Systems Analysis and Design*):
  - In November 1993 Alexander Stepanov presented a library based on generic programming to the ANSI/ISO committee for C++ standardization.
  - The committee's response was overwhelmingly favorable and led to a request from Andrew Koenig for a formal proposal in time for the March 1994 meeting.

# Standard Template Library

- The prospects for early widespread dissemination of the STL were considerably improved with Hewlett-Packard's decision to make its implementation freely available on the Internet in August 1994.
- This implementation, developed by Stepanov, Lee, and Musser during the standardization process, became the basis of many implementations offered by compiler and library vendors today.
- It provides four components called algorithms, containers, functions, and iterators.

# A “Must”

- Design and Analysis of Computer Algorithms is a must of Computer Curricula.
- In particular, it covers algorithm design patterns like *greedy method*, *divide-and-conquer*, *dynamic programming*, *backtracking* and *branch-and-bound*.

# Undergraduate vs. Graduate levels

- All listed design patterns are taught, learned and comprehended by examples.
- It is acceptable at *undergraduate* level. But is it a valid educating approach at *graduate* level?

# Glory of the Past

- Greedy Method
  - Continues Knapsack problem
  - Kruskal's algorithm
  - Huffman coding
- Backtracking
  - N-Queen Problem
  - Discrete Knapsack problem
  - Davis–Putnam–Logemann–Loveland algorithm

# More Glory …

- Branch & Bound
  - Discrete Knapsack problem
  - Interval Methods for Global Optimization
  - *Towards a Tractable Exact Test for Global Multiprocessor Fixed Priority Scheduling* (Artem Burnyakov talks for STEP-2024)
- Dynamic Programming
  - Dijkstra (shortest path) algorithm
  - Floyd–Warshall (shortest paths) algorithm
  - Cocke – Younger – Kasami algorithm

# Formalize!

- But these patterns can be formalized as design templates, rigorously specified, and mathematically verified.
- Greedy method is the only pattern that had been studied and formalized from rigor mathematical point of view in XX century.

# Formalization (?) of Greedy Method

- Edmonds J. Matroids and the greedy algorithm. Mathematical Programming, 1971, v.1, p.127-113.
- Korte B., Lovasz L. Mathematical structures underlying greedy algorithms. Lecture Notes in Computer Science, 1981, v.117, p.205-209.
- 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.274-283.

# Formalize!

- The speaker (i.t., me;-) published (in years 2010-2016) formalization, specification and (manual) verification for the following individual design techniques – backtracking, branch & bound, dynamic programming.
- In the present talk: survey of these formalizations from programming theory perspective and then … a move to a uniform/unified pattern in terms op $map$ and $reduce$.

# Talk outlines

- Introduction
- Levels of reasoning – interpreted and uninterpreted
- Recursive and iterative Dynamic Programming (DP)
- Recall about Backtracking (BT) and Branch & Bound (B&B)
- Unifying patterns for BT and B&B
- Conclusion (greedy, divide and conquer?)

Part 2

# LEVELS OF REASONING – INTERPRETED AND UNINTERPRETED

# Let’s start with recursion elimination

- A classic example of monadic recursion elimination (using reduction to the tail recursion) is function $M_{91}: \boldsymbol{N} \to \boldsymbol{N}$

$$M_{91}(n) = if\ n > 100\ then\ (n - 10)\ else\ M_{91}\big(M_{91}(n + 11)\big).$$

- It was introduced by John McCarthy, studied by Zohar Manna, Amir Pnueli, Donald Knuth. It turns out that

$$M_{91}(n) = if\ n > 101\ then\ (n - 10)\ else\ 91.$$

# Problem via recursion elimination

- A "key" idea elimination is a move from a monadic function $M_{91}: \boldsymbol{N} \to \boldsymbol{N}$ to a binary function $M2: \boldsymbol{N} \times \boldsymbol{N} \to \boldsymbol{N}$ such that $M2(n,k) = (M_{91})^k(n)$ for all $n, k \in \boldsymbol{N}$:

$$M2(n,k) = if\ k = 0\ then\ n$$
$$else\ if\ n > 100\ then\ M2\big((n-10),(k-1)\big)$$
$$else\ M2\big((n+11),(k+1)\big).$$

# Recursive factorial

- Recursive program to compute the factorial function $F: \boldsymbol{N} \rightarrow \boldsymbol{N}$
  - $F(n) = if\ n = 0\ then\ 1\ esle\ n \cdot F(n-1)$ (in the standard notation),
  - $F(n) = if\ p(n)\ then\ c\ else\ f\left(n, F\big(g(n)\big)\right)$ (in a prefix notation),

where *known* functions are

  - $p \equiv \big(\lambda\, x \in \boldsymbol{N}.\, (x = 0)\big) : \boldsymbol{N} \rightarrow Boolean,$
  - $c \equiv 1 :\rightarrow \boldsymbol{N}$ (i.e., a constant)
  - $f \equiv \big(\lambda\, x, y \in \boldsymbol{N}.\, (x \cdot y)\big) : \boldsymbol{N} \times \boldsymbol{N} \rightarrow \boldsymbol{N},$
  - $g \equiv \left(\lambda\, x \in \boldsymbol{N}.\big(if\ x = 0\ then\ 0\ else (x-1)\big)\right) : \boldsymbol{N} \rightarrow \boldsymbol{N}.$

# Imperative factorial

## Program 1

$VAR\ x, y: \boldsymbol{N};$

$y := 1;$

$while\ x \neq 0\ do$

$\quad y := x \cdot y;$

$\quad x := x - 1$

$od$

## Program 2

$VAR\ x, y, z: \boldsymbol{N};$

$y := 1; z \coloneqq 1;$

$while\ z \leq x\ do$

$\quad y \coloneqq z \cdot y;$

$\quad z \coloneqq z + 1$

$od$

# What if *known* functions are *uninterpreted*?

| Recursive schemata with a single available (not specified) data type $\boldsymbol{T}$: | |
|---|---|
| $F(x) = if\ p(x)\ then\ c\ else\ f\left(x, F\left(g(x)\right)\right)$ | |
| Standard scheme 1 | Standard scheme 2 |
| 1. $VAR\ x, y: \boldsymbol{T}$;<br>2. $y := c$;<br>3. $while\ \neg p(x)\ do$<br>4. $\quad y := f(x, y)$;<br>5. $\quad x := g(x)$<br>6. $od$ | 1. $VAR\ x, y, z: \boldsymbol{T}$;<br>2. $y := c; z := c$;<br>3. $while\ q(x, z)\ do$<br>4. $\quad y := f(z, y)$;<br>5. $\quad z := h(z)$<br>6. $od$ |

# Herbrand models and structures

- To demonstrate that not any two of program schemata from the previous slide are equivalent, it is sufficient to consider *Herbrand models* (also called *free models*).
- The domain of a Herbrand model comprises all terms constructed from the available functional symbols and input variables (while the domain of the Herbrand structures comprise the ground terms exclusively).

# Why the schemata aren't equivalent?

- Let us consider a Herbrand model such that
  - $q$ is always $TRUE$,
  - $p\left(g\big(g(x)\big)\right)$ is $TRUE$ while $p$ is $FALSE$ for all other terms.
- Then
  - $F(x) = f\left(x, F\big(g(x)\big)\right) = f\left(x, f\left(g(x), F\left(g\big(g(x)\big)\right)\right)\right) =$
    $= f\big(x, f(g(x), c)\big),$
  - the output value of $y$ computed by scheme 1 is $f\big(g(x), f(x, c)\big)$,
  - while scheme 2 does not halt at all.

# Translation of the recursive scheme to a standard scheme (with equality)

1. $VAR\ x, y, u, v : \boldsymbol{T};$
2. $u := x;$
3. $while\ \neg p(u)\ do$
4. $\quad u := g(u)$
5. $od$
6. $y := c;$
7. $while\ u \neq x\ do$
8. $\quad v := x;$
9. $\quad while\ g(v) \neq u\ do$

$\qquad Inv.1\text{: } \exists m < n \in \boldsymbol{N} :\ v = g^m(x)\ \&\ u = g^n(x)$

$\qquad v := g(v)$

$\quad od;$

$\quad Inv.2\text{: } g(v) = u\ \&\ y = F(u)$

10. $\quad y := f(u, y);\ u := v$
11. $od;$
12. $y := if\ p(x)\ then\ c\ else\ f(x, y)$

# How to rid of the equality

- Finally, the equality used in lines 7 and 9 of the scheme is easy to eliminate because it may be implemented as call of the following *tail-recursive* function $EQ$ (easy to implement by an iterative program:

```
1   VAR x, y, u, v : D;
2   u := x;
3   while ¬p(u) do
4       u := g(u)
5   od
6   y := c;
7   while u ≠ x do
8       v := x;
9       while g(v) ≠ u do
            //Invariant 1: ∃m < n ∈ ℕ : v = g^m(x) & u = g^n(x)
            v := g(v)
        od;
        //Invariant 2: g(v) = u & y = F(u)
10      y := f(u, y); u := v
11  od
```

$$EQ(a,b) = \ if\ p(a) \vee p(a)\ then\ p(a)\ \&\ p(b)\ else\ EQ\big(g(a), g(b)\big).$$

# Translation of the recursive factorial to an iterative form

1. $VAR\ x, y, u, v : \boldsymbol{N};$
2. $u := x;$
3. $while\ u \neq 0\ do$
4. $\quad u := u - 1$
5. $od$
6. $y := 1;$
7. $while\ u \neq x\ do$
8. $\quad v := x;$
9. $\quad while\ (v - 1) \neq u\ do$

$\qquad$ <u>$Inv.1: \exists m < n \in \boldsymbol{N} : v = x - m \ \&\ u = x - n$</u>

$\qquad v := v - 1$

$\quad od;$

$\quad$ <u>$Inv.2:\ (v - 1) = u \ \&\ y = F(u)$</u>

10. $\quad y := u \cdot y;\ u := v$
11. $od;$
12. $y := if\ (x = 0)\ then\ 1\ else\ (x \cdot y)$

# Extremely inefficient but semantic-independent

- Unfortunately, imperative factorial from the previous slide 10 is extremely inefficient – it runs in $O(n^2)$ time in contrast to both programs (1 and 2) from slide 4 that run in linear time $O(n)$.
- It worth to remark that Program 1 can be automatically constructed from the recursive factorial program using *co-recursion* and then *tail-recursion.*
- This use of the co-recursion is semantic-dependent (since it is safe assuming commutativity of the function $f$), while our approach to recursion elimination is semantic-independent.

# Co-recursion and Tail-recursion by example

- Recursive factorial $F(n) = if\ n = 0\ then\ 1\ esle\ n \cdot F(n-1)$ is not in the tail-form (because has next call inside some function).
- But it is equivalent to the following recursive program in the tail-form:

$$\begin{cases} F(n) = P(n,1) \\ P(n,m) = if\ n = 0\ then\ m\ esle\ P\big((n-1),(n \cdot m)\big) \end{cases}.$$

- This program is in the tail-form because all calls are never inside other functions.
- Co-recursion is a “trick” that consists in converts result into another argument and use this argument in the recursion.

# Teil-recursion elimination by example

- Tail-recursion $\begin{cases} F(n) = P(n,1) \\ P(n,m) = if\ n = 0\ then\ m\ esle\ P\big((n-1),(n \cdot m)\big) \end{cases}$

  is easy to eliminate (and compare with Program 1 from slide 4):

| | |
|---|---|
| $\underline{start}$: $VAR\ x, y$: $\boldsymbol{N}$ $goto\ \underline{2}$<br>$\underline{2}$: $y := 1\ goto\ \underline{3}$<br>$\underline{3}$: $if\ x = 0\ then\ goto\ \underline{stop}\ else\ goto\ \underline{4}$<br>$\underline{4}$: $y := x \cdot y\ goto\ \underline{5}$<br>$\underline{5}$: $x := x - 1\ goto\ \underline{3}$<br>stop | 1. $VAR\ x, y: \boldsymbol{N}$;<br>2. $y := 1$;<br>3. $while\ x \neq 0\ do$<br>4. $y := x \cdot y$;<br>5. $x := x - 1$<br>6. $od$ |

Part 3

# RECURSIVE AND ITERATIVE DYNAMIC PROGRAMMING

# Warming-up Dropping Bricks Problem

- Define stability of “bricks” (cell phones) by dropping them from a tower of H meters. How many times do you need to drop bricks, if you have just 2 bricks?
- $G(n) = if\ n = 0\ then\ 0\ else$

$$1 + \min_{1 \le k \le n} \max\{(k-1), G(n-k)\}.$$

# History of “Dynamic Programming”

- *Dynamic Programming* was introduced by Richard Bellman in the 1950s to tackle optimal planning problems.
- In 1950s 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 states* (compare – *dynamic logic*, *dynamic system*).

# Bellman equation and optimality principle

- *Bellman equation* is a functional equality for the objective function that expresses the optimal solution at the *current* state in terms of the optimal solution at *next* (changed) states.
- It is conceptualized a so-called *Bellman Principle of Optimality*: an optimal plan (or program) should be optimal at every stage.

# Descending (top-down) Dynamic Programming

- General pattern of Bellman equation may be formalised by the following *scheme of recursive descending Dynamic Programming*:

$$G(x) = if\ p(x)\ then\ f(x)\ else$$

$$\underbrace{g\left(x, \left\{h_i\left(x, G(t_i(x))\right) : i \in [1..n(x)]\right\}\right)}_{\text{the term is linear in each branch w.r.t. the objective function G}};$$

the term is *linear in each branch* w.r.t. the objective function G

# Descending (top-down) Dynamic Programming – cont.

- In this scheme
  - $G: X \to Y$ is a symbol for the objective function,
  - $p$: $X \to Bool$ is a symbol for a known predicate,
  - $f: X \to Y$ is a symbol for a known function,
  - is a symbol for a known function with a variable (but finite) number of arguments,
  – all $h_i: X \times Z \to Y$, $i \in [1..n(x)]$ are symbols for known functions,
  – all $h_i: X \to X$, $i \in [1..n(x)]$ are symbols for known functions too.

# More Examples: Factorial, Fibonacci Numbers and Words

- $F(n) = if\ n = 0\ then\ 1\ else\ n \cdot F(n-1);$
- $Fib(n) = if\ 0 \leq n \leq 1\ then\ 1\ else\ Fib(n-2) + Fib(n-1);$
- $Wrd(n) = if\ n = 0\ then\ a$

  $else\ if\ n = 1\ then\ b$

  $else\ Wrd(n-2) \circ Wrd(n-1).$

# Observations

- Factorial, Fibonacci Numbers and Words need static memory of a fixed size.
- Surprisingly, but Dropping Bricks Problem also needs just static memory of fix-size, since $G(n) = \arg\min k \in \boldsymbol{N}\colon \left(\frac{k(k+1)}{2} \geq n\right)$.

# Research questions about Descending Dynamic Programming

- It follows from Paterson M.S. and Hewitt C.T. paper *Comparative Schematology* (1970) that fix-size *static memory* is *not enough* for recursion elimination in Bellman equation.
- When one-time allocated
  - array (with integer indexes),
  - (fix-size) static memory

  is sufficient to eliminate recursion in Bellman equation?

# A need of dynamic (size) memory

- The following program scheme

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

is not equivalent to any standard program scheme:

for every $n > 0$

there exists an Herbrand model $T_n$

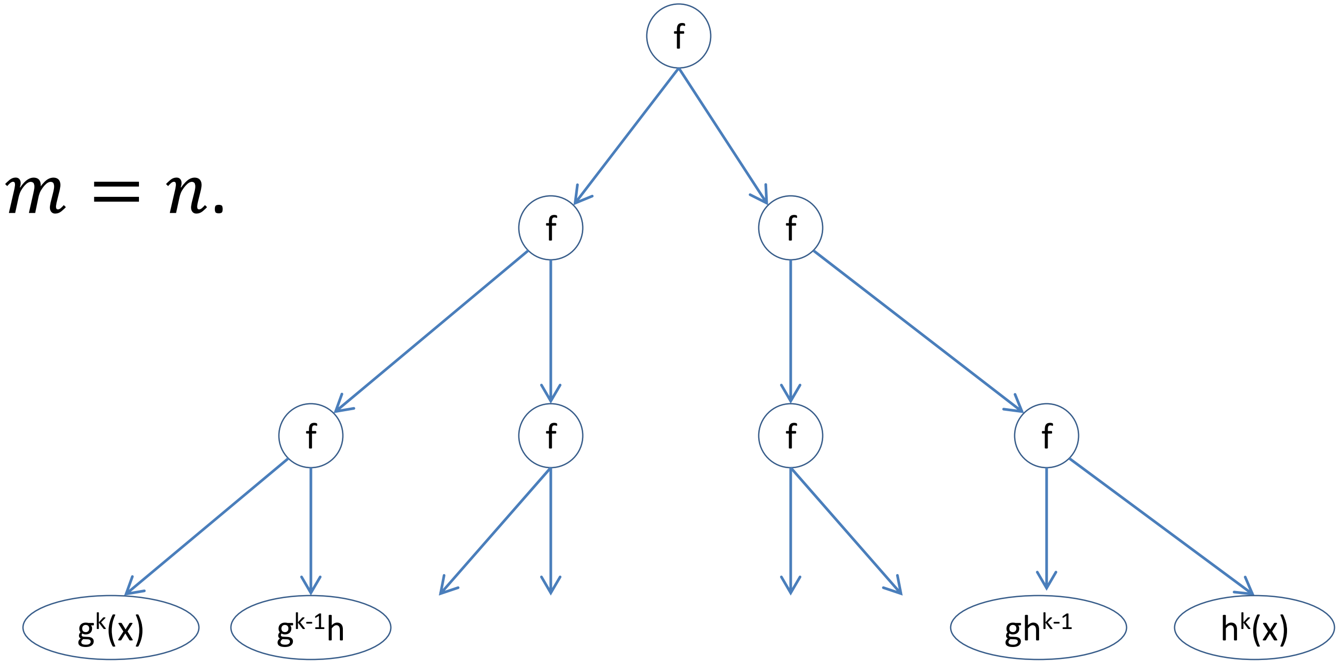
where any standard program scheme

needs $n$ variables to compute $F$.

# A need of dynamic (size) memory (proof idea)

Consider the following data type $T_n$:

- values are sub-terms of the term $t_n$ depicted to the right;
- $p\left(g^k\left(h^m(x)\right)\right)$ is true, if $k + m = n$.



# A need of dynamic (size) memory (proof idea)

- Observe that if $F(x) \; = \; if \; p(x) \; then \; x \; else \; f\left(F\big(g(x)\big), F\big(h(x)\big)\right)$ then $F(x) = t_n$.
- Prove by induction: any iterative algorithm that computes $t_n$ needs $n$ variables (memory cells) at least.

# Support of the Objective Function

- If $G(x) = if\ p(x)\ then\ f(x)\ else$
$$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in [1..n(x)]\right\}\right)$$
is defined for some value $v$, then it is possible to pre-compute the *support* $\mathrm{spp}(v)$, the set of all values that occur in the computation of $G(v)$:
$$\mathrm{spp}(x) = if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup \left(\bigcup_{i \in [1..n(x)]} \mathrm{spp}\left(t_i(x)\right)\right).$$
- Remark, that for every $v$, if $G(v)$ is defined, then $\mathrm{spp}(v)$ is finite (but not vice versa).

# When an array suffices

- One-time allocated array with integer indexes suffices for computing $G(x) = if\ p(x)\ then\ f(x)\ else$
$$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in \left(1..n(x)\right)\right\}\right)$$
  if $n$ is a constant and all $t_i$, $i \in \left(1..n(x)\right)$, are interpreted by commutative functions.

# When static memory suffices

- Fix-size static memory suffice for computing
  $G(x) = if\ p(x)\ then\ f(x)\ else$
  $$g\left(x, \left\{h_i\left(x, G\big(t_i(x)\big)\right) : i \in \big(1..n(x)\big)\right\}\right)$$
  if $n(x) = n$ is a constant and there exists a known computable function $t$ such that
  - $t_i = t^i$ for all $i \in [1..n]$,
  - $p(u)$ implies $p\big(t(u)\big)$ for all $u \in \mathrm{spp}(x)$.
- Examples: Factorial, Fibonacci Numbers and Words.
- Counter-example: Paterson-Hewitt scheme.

# Design outlines and proof comments

## Proof comments

- Proof idea – very same as for factorial function in Part 1.
- Scheme' design (with equality and invertible function $t$) is depicted to the right.

## Design outlines

```
VAR x, x1, ... xn : X;
VAR y, y1, ... yn : Y;
x := v;
if p(x) then y := f(x)
    else { do x := t1(x) until p(x);
           x1 := x; y1 := f(x1);
           x2 := t(x1); y2 := f(x2);
           ... ...
           xn := t(xn-1); yn := f(xn);
           do
               x := t⁻(x);
//Invariant: x = t⁻(x1) & bas(x) = {x1, ... xn} &
//Invariant: & y1 = G(x1) & ... & yn = G(xn)
               y := g(x, (h1(x, y1), ... hn(x, yn)));
               yn := yn-1; ... y3 := y2; y2 := y1;
               y1 := y;
               x1 := t⁻(x1); ... xn := t⁻(xn)
           until x = v }.
```

# Selected references

1. G. Berry. Bottom-up computation of recursive programs. RAIRO | Informatique Th´eorique et Applications (Theoretical Informatics and Applications, 10(3):47-82, 1976.
2. R. S. Bird. Zippy tabulations of recursive functions. In Proceedings of the 9th International Conference on Mathematics of Program Construction, MPC ’08, pages 92-109. Springer-Verlag, 2008.
3. J. Cowles and R. Gamboa. Contributions to the theory of tail recursive functions, 2004. Available at http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf.
4. D.E. Knuth. Textbook examples of recursion. arXiv:cs/9301113[cs.CC], 1991.
5. Y. A. Liu. Systematic Program Design: From Clarity to Efficiency. Cambridge University Press, 2013.
6. M.S. Paterson and C.T. Hewitt. Comperative schematology. In Proc. of the ACM Conf. on Concurrent Systems and Parallel Computation, pages 119-127. Association for Computing Machinery, 1970.
7. N.V. Shilov, D. Danko Teaching Efficient Recursive Programming and Recursion Elimination Using Olympiads and Contests Problems. In Proc. of the workshop on Frontiers in Software Engineering Education (FISEE-2019), Lecture Notes in Computer Science, 2020, v.12271, p.246-264.

Part 3

# RECALL ABOUT BACKTRACKING AND BRANCH & BOUND

# Two slides that were dropped on May 22: Support of the Objective Function

- If $G(x) = if\ p(x)\ then\ f(x)\ else$
$$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in [1..n(x)]\right\}\right)$$
is defined for some value $v$, then it is “possible” to pre-compute the *support* $\mathrm{spp}(v)$ – the set of all values that occur while computing $G(v)$:
$$\mathrm{spp}(x) = if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup \left(\bigcup_{i\in[1..n(x)]} \mathrm{spp}\left(t_i(x)\right)\right).$$
- Remark, that for every $v$, if $G(v)$ is defined, then $\mathrm{spp}(v)$ is finite (but not vice versa).

# Two slides that were dropped on May 22: When an array suffices

- One-time allocated array with integer indexes suffices for computing $G(x) = if\ p(x)\ then\ f(x)\ else$
$$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in \left(1..n(x)\right)\right\}\right)$$
if $n$ is a constant and all $t_i$, $i \in \left(1..n(x)\right)$, are interpreted by commutative functions.

# Back to Dropping Bricks Problem

- Unfortunately, the techniques developed above lead to use of
  - a (one time allocated) array,
  - but not a fix-size static memory…

# Dynamic Programming for knapsack with undividable goods

- Knapsack problem for undividable goods can be formulated in the form of descending dynamic programming,

but

- when gross capacity and/or individual weights are real values the computation of the support function $spp$ has the same complexity as the problem itself!

# Dynamic Programming for knapsack with undividable goods – cont.

- For example, the maximal price of $n \geq 0$ undividable goods $(W_1, P_1)$, … $(W_n, P_n)$ that may be accumulated in a knapsack with capacity $W$ may be computed recursive algorithm (that match dynamic programming pattern):

$$MaxPrice(W, n) = if\ n = 0\ then\ 0\ else$$

$$if\ W_n > W\ then\ MaxPrice\big(W, (n-1)\big)\ else$$

$$\max\big\{MaxPrice\big(W, (n-1)\big), P_n + MaxPrice\big((W - W_n), (n-1)\big)\big\}$$

# Graph Traversals

- In cases like knapsack with undividable goods, it remains to *travers* the *decision* tree (i.e., the tree of recursive calls) of the problem using *backtracking* or *branch-and-bound* methodology.
- In general, graph traversal refers to the problem of visiting all the nodes in a (di)graph to compute some graph characteristics (in particular, to find any/all nodes/vertices that enjoy some property specified by some Boolean *criterion condition*).

# Back to descending Dynamic Programming

- Bellman equation is already a functioal program and

  $$G(x) = if\ p(x)\ then\ f(x)\ else$$

  $$g\left(x, \left\{h_i\left(x, G\left(t_i(x)\right)\right) : i \in [1..n(x)]\right\}\right)$$

  and its computations may be considered as a traversal of the tree of recursive calls.

# Depth- and Breadth-first Traversals

- 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 graph that visits the sibling nodes before visiting the child nodes.

# Backtracking and Branch-and-Bound

- 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
# Branch-and-Bound and Backtracking

- Branch-and-bound (B&B) is DFS that uses boundary condition, the method was introduced in the paper

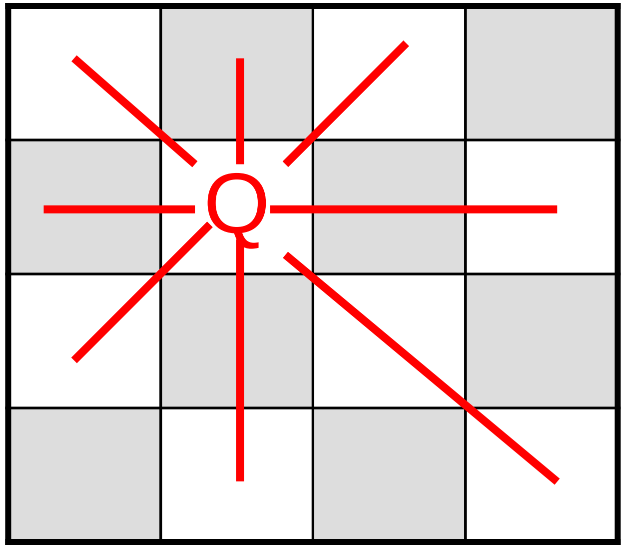
  Land A. H. and Doig A. G. An automatic method of solving discrete programming problems. Econometrica, 28(3), 1960, p.497-520.

- Backtracking (BT) is DFS that uses boundary condition, the method was introduced in the paper

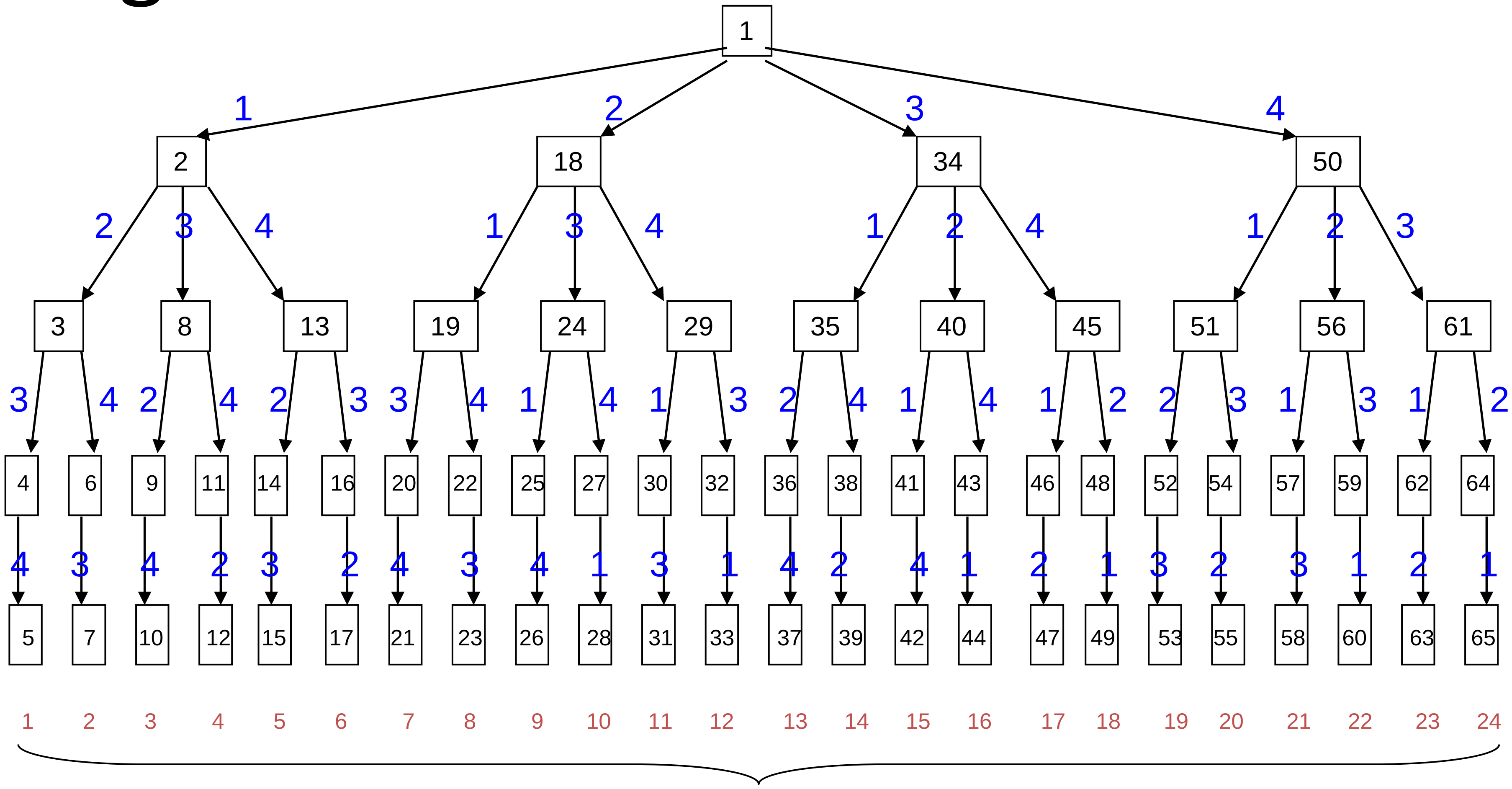
  Golomb S.W. and Baumert L.D. Backtrack Programming. Journal of ACM, 12(4), 1965, p.516-524.

# Example: Four Queens Puzzle

- Place $4$ queens on simplified $4 \times 4$ chessboard so that non attacks another (*criterion condition*).
- Naïve Algorithm:
  - generate *ALL* possible placements proceeding row by row, and square by square in the row;
  - try criterion condition for each generated placement.



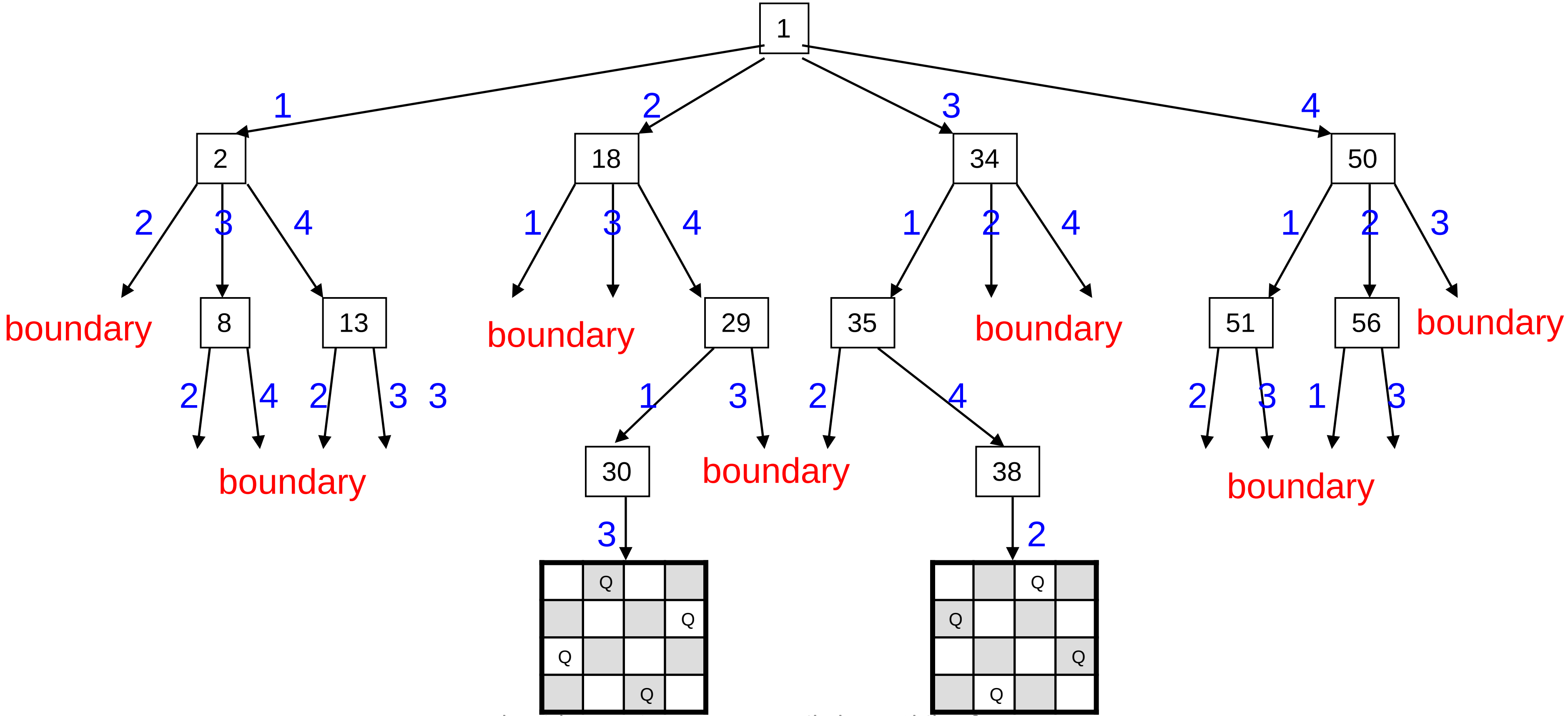
# Four Queens Puzzle: Naïve Algorithm



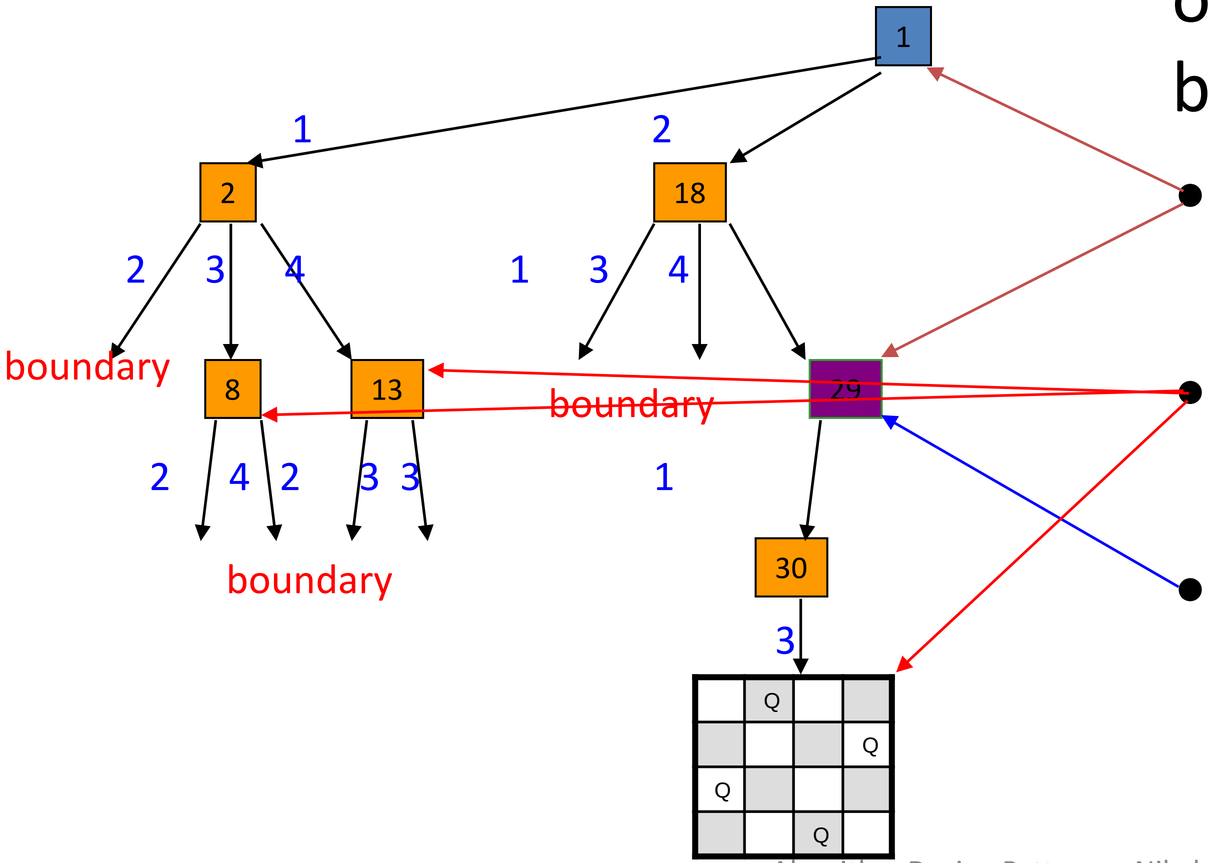

indexes for positioning

# Four Queens Puzzle: Backtracking and Branch-and-Bound

Some subtrees may be refuted on the fly due to *boundary condition*: some queens attack each other



# Basic Terminology




Nodes that emerge in the process of traversing (of a tree with some boundary condition) can be

- *live* (some of its children are not generated yet),
- *dead* (all its children has been generated),
- *expanding* (currently processing).

# Interval Method for Global Optimization

- Most global optimization methods using *interval techniques* employ a branch-and-bound strategy:

  Gray P., HartW., Painton L., Phillips C., Trahan M.,Wagner J. A Survey of Global Optimization Methods. Sandia National Laboratories, 1997 (http://www.cs.sandia.gov/opt/survey/main.html).
- 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 interval technique.

Part 4

# UNIFYING TEMPLATES FOR BT AND B&B

# Temporal ADT Teque

- *Theque* is a finite set of values (of some background data type) marked by disjoint *time-stamps*.
- The time stamps are readings of a *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 $t$ by the pair $(x, t)$. Readings of the clock as well as time-stamps are not visible for any observer.

# Temporal ADT Teque – cont.

- Theque inherits some set-theoretic operations: the empteq (i.e., *empty teque*) is simply the empty set ($\emptyset$), set-theoretic equality ($=$) and inequality ($\neq$), subset relations (for example, $\subseteq$).
- ADT theque has its own specific operations, some of these operations are *time-independent*, some others are *time-sensitive*, and some are *time-dependent*.

# Time-independent operations

- Operation $Set$: for every teque $T$ let $Set(T)$ be $\{x \mid \exists t:\ (x, t) \in T\}$.
- Operations $In$ and $Ni$: for every teque $T$ and any value $x$ (of the background type)
  - let $In(x, T)$ stay for $x \in Set(T)$,
  - and $Ni(x, T)$ stay for $x \notin Set(T)$.
- Operation $Spec$ (specification): for every teque $T$ and any predicate $\lambda x: Q(x)$ on values of the background type let teque $Spec(T, Q)$ be the following sub-teque $\{(x, t) \in T : Q(x)\}$.

# Time-dependent operation *AddTo*

- For every list of teques $T_1$,… $T_n$ ($n \geq 1$) and
  any finite set $\{x_1, \dots x_m\}$ of elements of the background type ($m \geq 0$),
  let execution $AddTo(\{x_1, \dots x_m\}, T_1, \dots T_n)$ at time $t$
  returns $n$ teques $T'_1$,… $T'_n$ such that
  for some moments of time $t = t_1 < \cdots < t_m = t'$
  (where $t'$ is the the *moment of termination* of the operation),
  $T'_i = T_i \cup \{(x_1, t_1), \dots (x_m, t_m)\}$ for all $i \in [1..n]$.

# Time-sensitive operations

- There are three pairs of time-sensitive operations:
  - $Fir$ and $RemFir$ (“head” and “tail”),
  - $Las$ and $RemLas$ (“top” and “pop”),
  - $Elm$ and $RemElm$ (“random” and “drop it”).
- Let $T$ be a teque.
  - Let $Fir(T)$ be the value of the background type 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).

# Time-sensitive operations – cont.

- Let $Las(T)$ be the value of the background type 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).
- Let $Elm(T)$ be some element (somehow defined or specified, even randomly) of $T$ (also without any time-stamp) and $RemElm(T)$ is the teque that results from $T$ after removal of this element (with its time-stamp).

# Notational convention

- Let pair of $FEL$ and $REM$ stays simultaneously for
  - either $Fir$ and $RemFir$,
  - or $Las$ and $RemLas$,
  - or $Elm$ 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.

# Teque Convention

- Instantiation of $Fir$ and $RemFir$ imposes queue discipline *first-in — first-out* and specializes the unified template to B&B template.
- Instantiation of $Las$ and $RemLas$ imposes stack discipline *first-in — last-out* and specializes the template to BT template.
- Instantiation of $Elm$ and $RemElm$ specializes the unified template to *Deep Backtracking*, *Branch and Bounds with priorities*, or even a *random walk* templates.

# Virtual Graph Notation

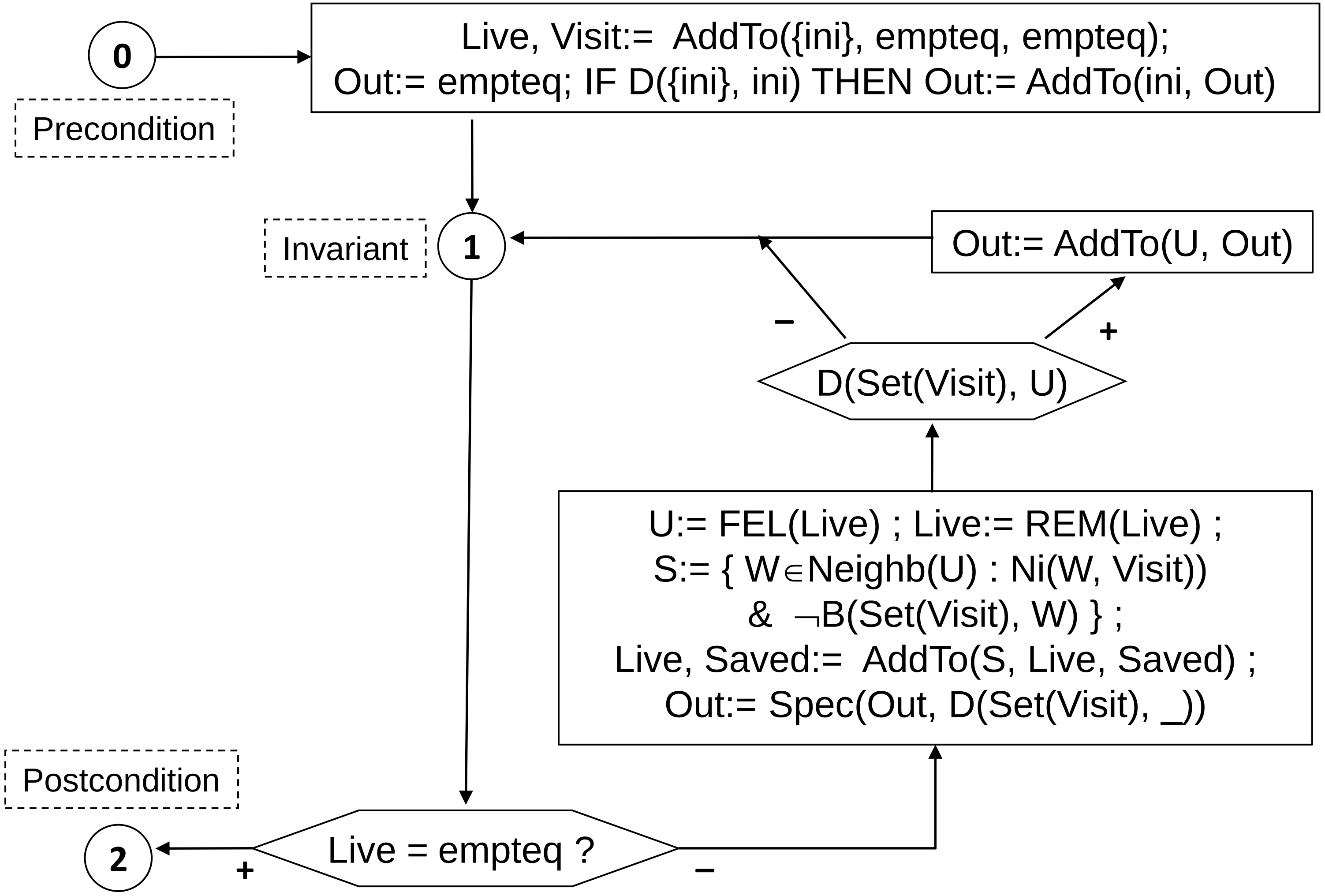
- A virtual (di)graph $G$ *is defined by means of* the following features:
  - a type $Node$ of vertices and the initial vertex $ini$ of this type such that every vertex of $G$ is reachable from $ini$;
  - a computable function $Neighb : Node \to 2^{Node}$ that for any vertex in $G$ returns the set of all its neighbors (children in a digraph).

# Boundary and Decision Conditions

- Let us introduce *easy to cheque*
  - a *Boundary* condition $B: 2^{Node} \times Node \to Boolean$
  - and a *Decision* condition $D: 2^{Node} \times Node \to Boolean$

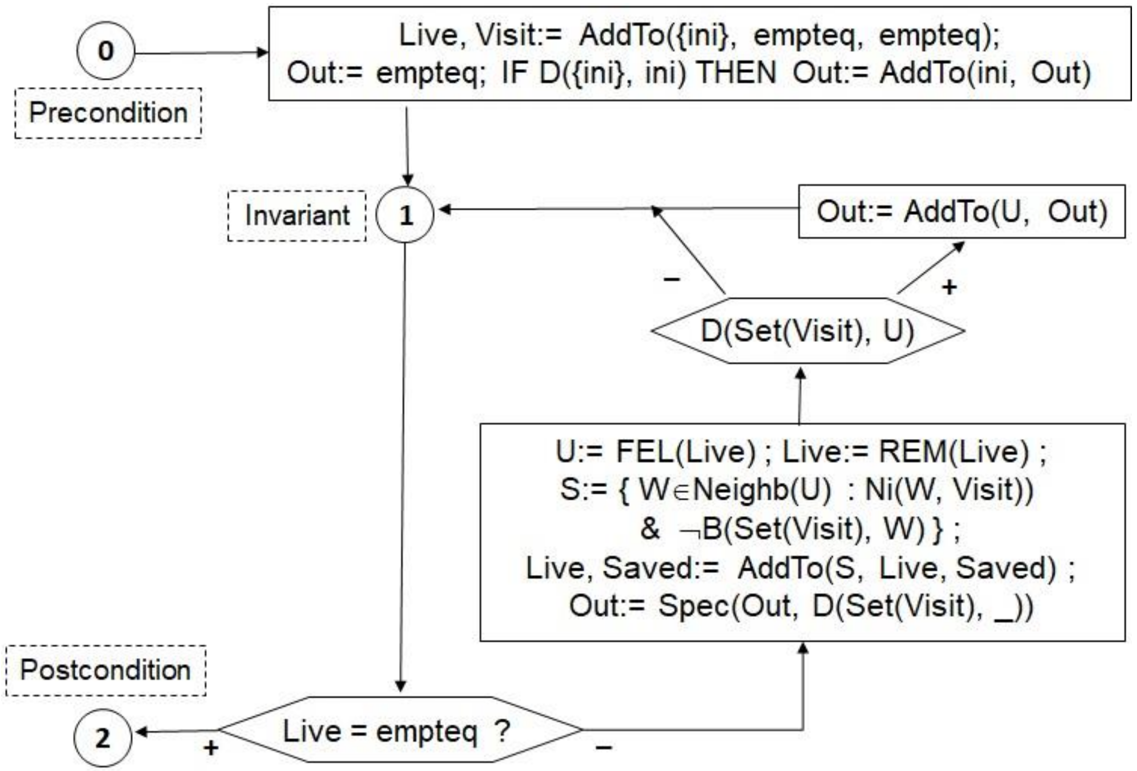
  to be used for collecting all nodes that meet a *hard to cheque Criterion* condition $C: Node \to Boolean.$

# Template


0
Live, Visit:= AddTo({ini}, empteq, empteq);
Out:= empteq; IF D({ini}, ini) THEN Out:= AddTo(ini, Out)
Precondition
Invariant
1
Out:= AddTo(U, Out)
–
+
D(Set(Visit), U)
U:= FEL(Live) ; Live:= REM(Live) ;
S:= { W∈Neighb(U) : Ni(W, Visit))
& ¬B(Set(Visit), W) } ;
Live, Saved:= AddTo(S, Live, Saved) ;
Out:= Spec(Out, D(Set(Visit), _))
Postcondition
2
+
Live = empteq ?
–

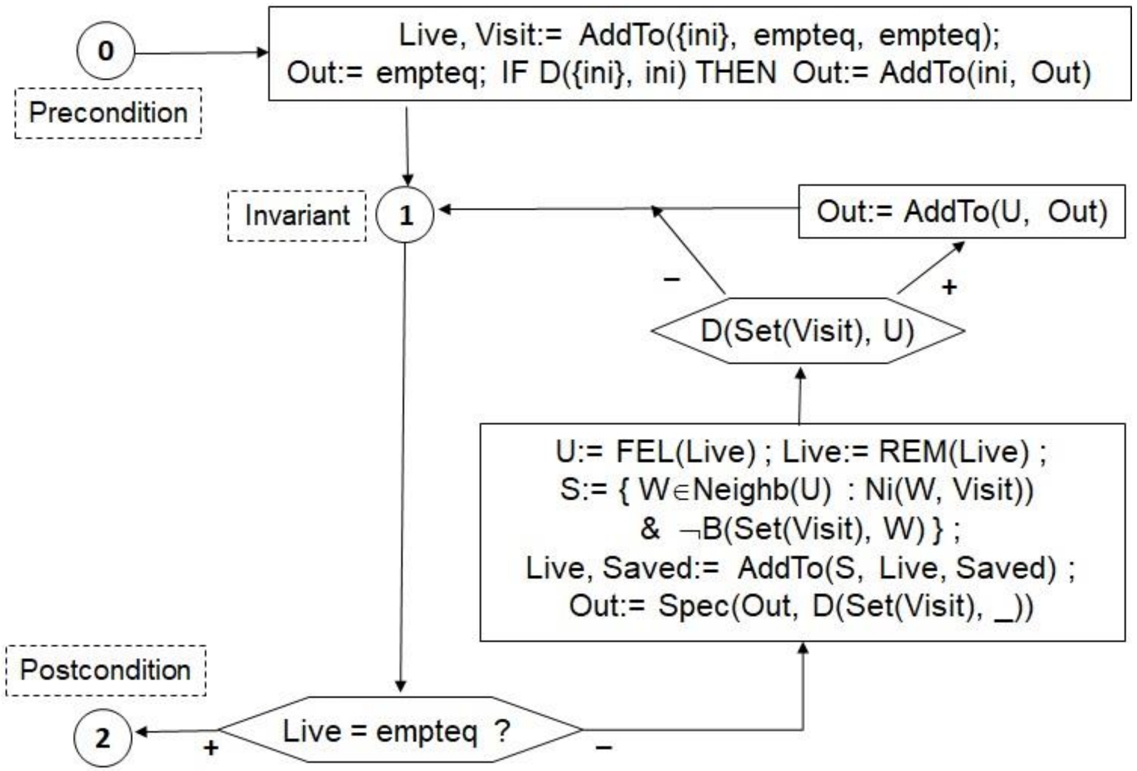
# Specification: Postcondition

Teque $Out$ consists (with time-stamps) of all nodes of the graph $G$ that meet the criterion condition $C$, and each of these nodes has single entry (occurrence) in $Out$.



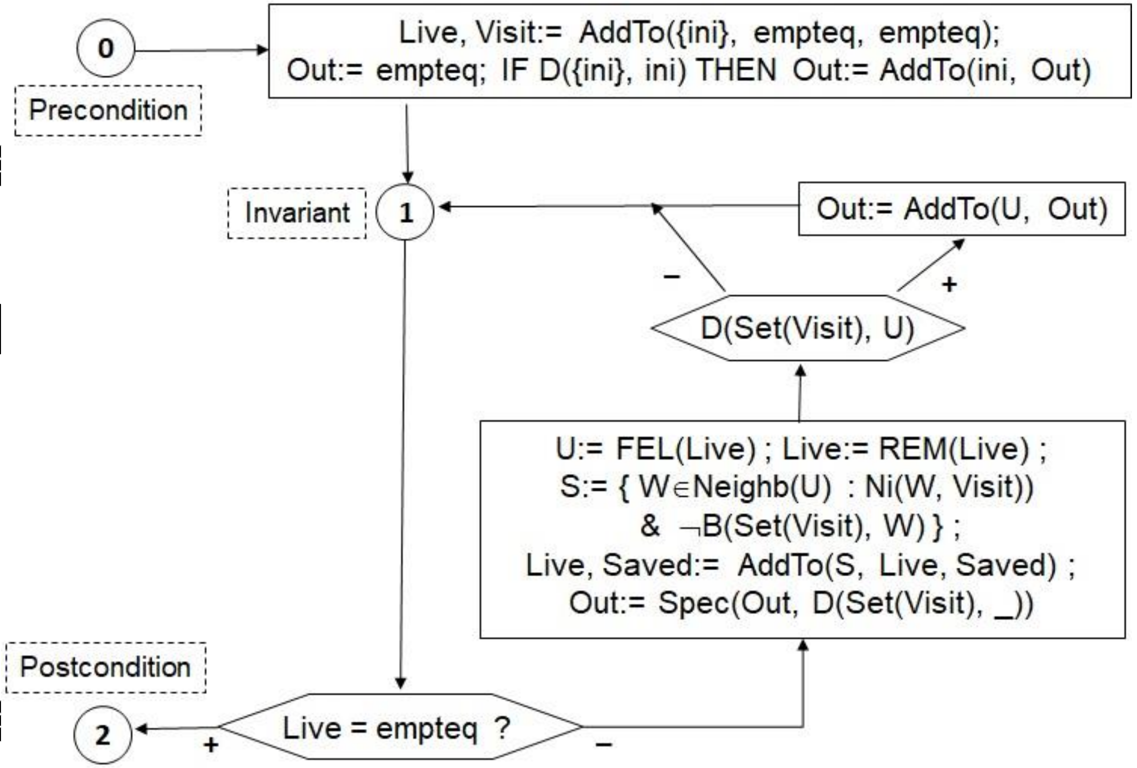
# Specification: Preconditions 1 and 2

1. A virtual (di)graph $G$ is defined by the initial node $ini$ and the neighborhood function $Neighb$.
2. For every node $x$ of $G$ the boundary condition $\lambda S{:}\,B(S,x)$ is a monotone function (i.e., if a node is ruled-out by a set, then it is ruled-out by any larger set).



# Specification: Preconditions 3 and 4

3. For every set $S$ of nodes of $G$ the decision condition $\lambda x: D(S, x)$ is a monotone function in the following sense: if a node is ruled-out by the set then all its successors are ruled out by the set also.
4. For every node $x$ of $G$ the decision condition $\lambda S: D(S, x)$ is an anti-monotone function in the following sense: a candidate node may be discarded later by a lager set.



# Specification: Precondition 5

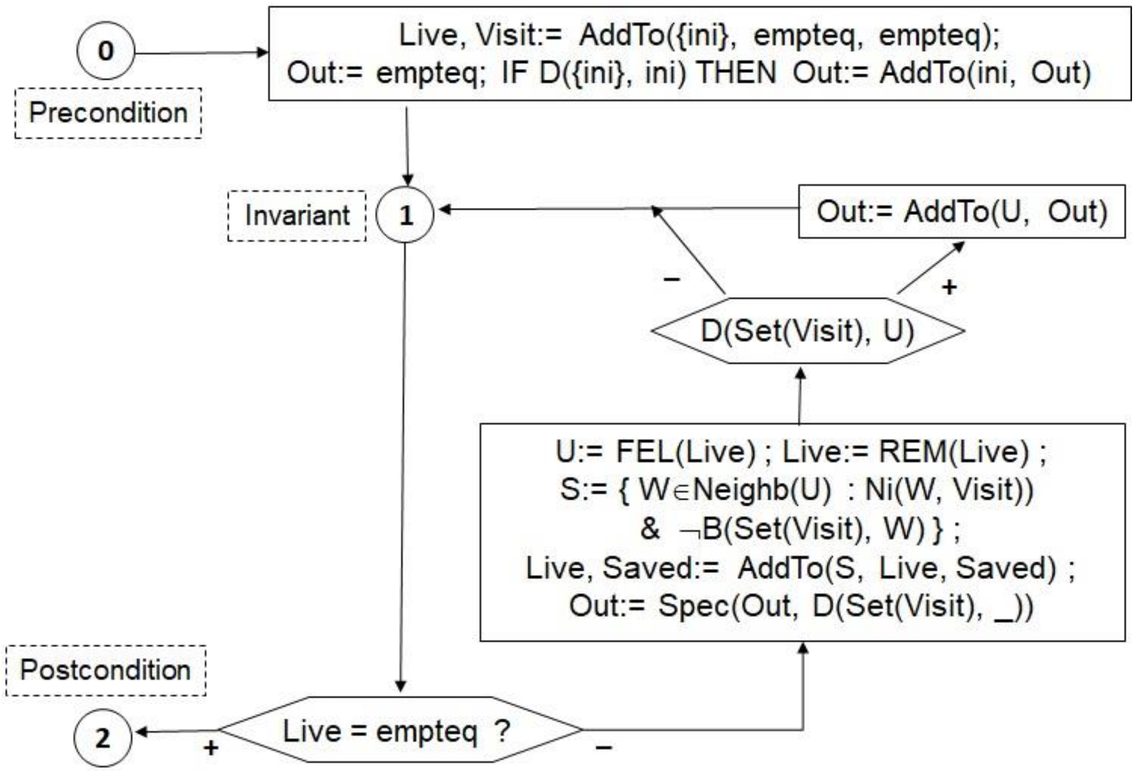
5. For every set of nodes $S$:

   if $S \cup \{x \in G : B(S, x)\} =$

   all nodes of $G$,

   then $D(S, x) \Leftrightarrow C(x)$

   (i.e., the decision condition $D$ applied to a set with *a complete extension* is equivalent to the criterion condition $C$).



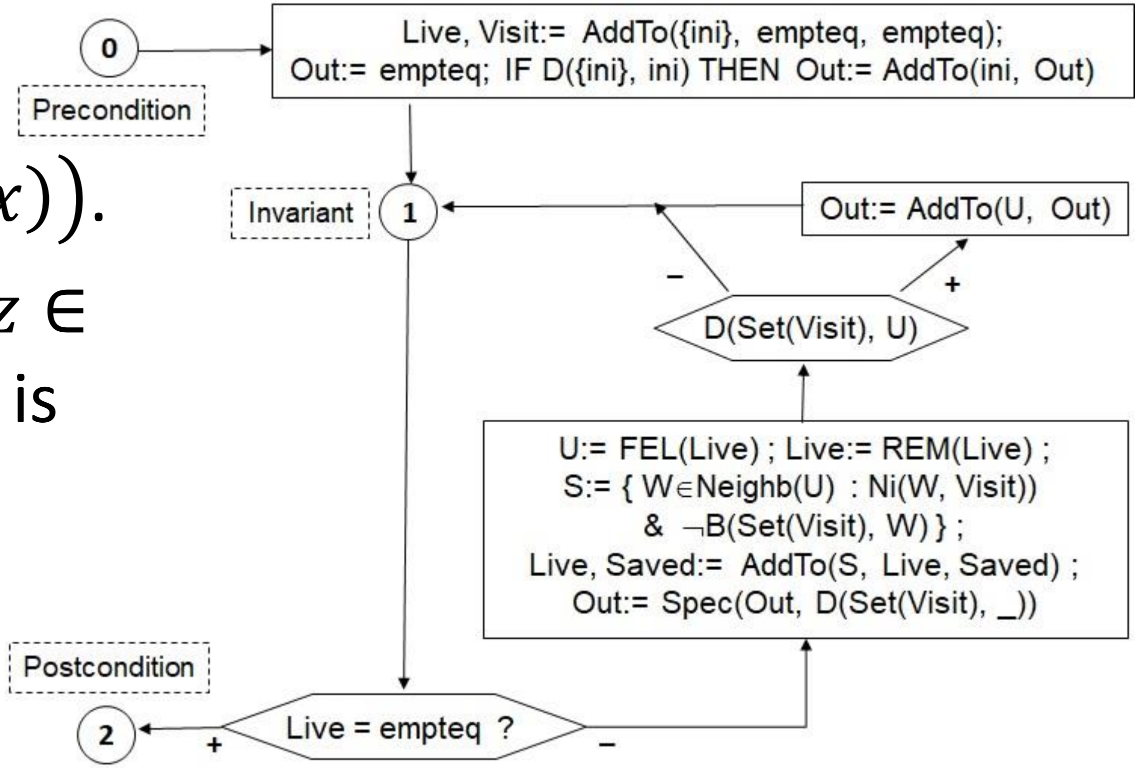
# Loop Invariant

Conjunction of the following 4 clauses:

1. $Out =$

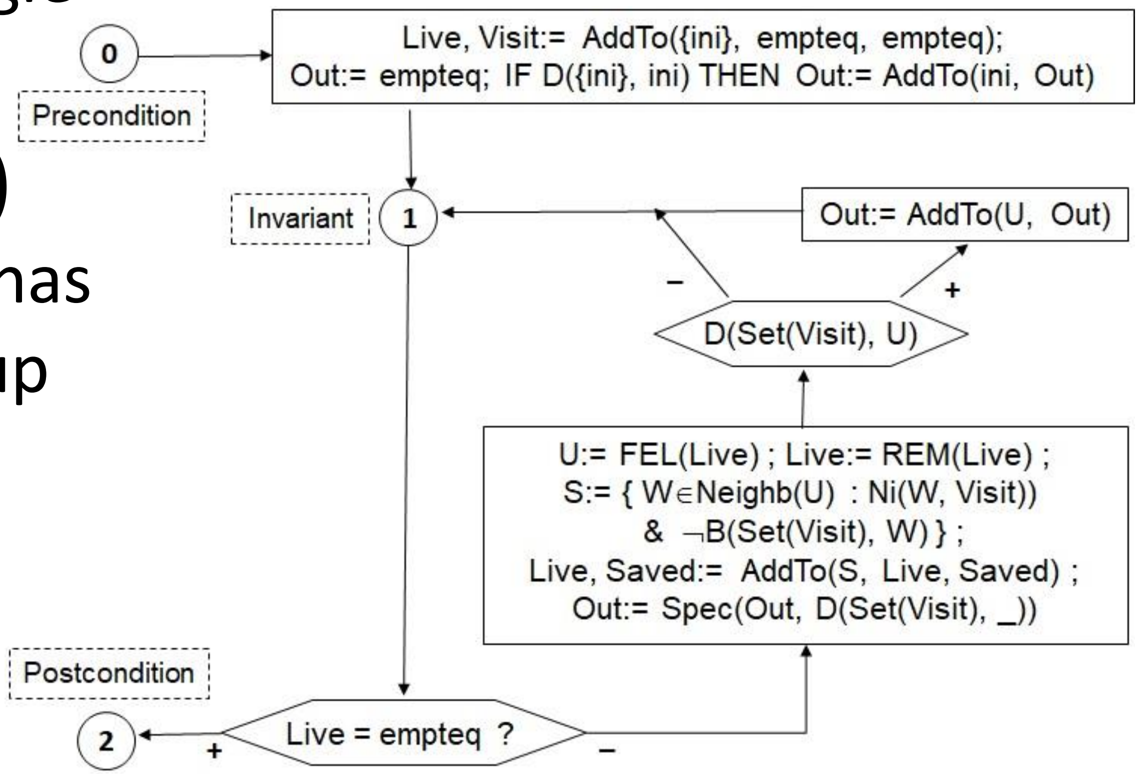
   $= Spec\big(Visit, \lambda x{:}\, D(Set(Visit), x)\big)$.

2. $Live \subseteq Visit$, and for every node $z \in G$, if $Ni(z, Visit)$ and $C(z)$, then $z$ is reachable from $Set(Live)$.



# Loop Invariant – cont.

3. Each node $x \in G$ has (at most) single instance in $Visit.$
4. $Set(Visit) \cup Neighb\big(Set(Visit)\big)$ equals to the set of all nodes that has been generated by the algorithm up to the current moment of time.



# Correctness

- If the boundary, decision and criterion conditions $B$, $D$ and $C$ meet the precondition, and the virtual graph $G$ for traversing is finite,
- then every algorithm instantiated from the template terminates after $O(|G|)$ iterations of the loop,
- and upon termination the final value of $Set(Out)$ is the set of all nodes of $G$ that meet the criterion condition $C$.

Part 5

# CONCLUDING REMARKS ON BT AND B&B DESIGN TEMPLATES

# BT and B&B

- We have discussed and present a unified template for BT and B&B algorithm design patterns,
- specified the template by means of (semiformal) precondition and postcondition,
- validate it manually by Floyd method.

# Directions for further research

- Formalization of the template and its specification, development of a computer-aided proof in some proof-assistant system.
- Study of the algorithm design templates from mixed computations perspective for automatic algorithm generation.
- Implementation as a template library in C++ to extend STL and try its efficiency (educational as well as practical).

# Templates:
# Mixed Computation Perspective

- 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 by design algorithms to solve concrete problems.
- The purpose is closely related to Mixed Computations and/or Partial Evaluation:
  - Ershov A.P. Mixed computation: potential applications and problems for study. Theor. Comp. Sci., 1982, v18(1), p.41-67.
  - Jones J.D., Gomard C.K., and Sestoft P. Partial Evaluation and Automatic Program Generation. Prentice Hall International, 1993.

# Templates: Mixed Computation Perspective

- The difference consists in level of consideration:
  - in our case we use algorithm design templates and use pseudo-code,
  - while in Mixed Computations and Partial Evaluation program code and programming languages are in use.

# Selected references

1. 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.
2. Silov N.V. Verification of Backtracking and Branch and Bound Design Templates. Automatic Control and Computer Sciences, 2012, v.46(7), p.402-409.
3. 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.
4. Shilov N.V. *Algorithm Design Patterns: Program Theory Perspective.* In Proceedings of Fifth International Valentin Turchin Workshop on Metacomputation, 2016, University of Pereslavl, p. 170-181.

# Thank you! Questions?