©Journal of the Technical University Sofia, branch Plovdiv
"Fundamental Sciences and Applications", Vol. 16, 2011
International Conference Engineering, Technologies and Systems
TechSys 2011
BULGARIA

DECLARATIVE AND PROCEDURAL SEARCH STRATEGY IMPLEMENTATIONS IN WINSPIDER

KOSTADIN KRATCHANOV[1], EMILIA GOLEMANOVA[2], TZANKO GOLEMANOV[2], YASEMIN GÖKÇEN[1]

Abstract. ***Control Network Programming (CNP) is a style of programming developed by the authors and other colleagues which combines and extends three major programming paradigms – imperative programming, declarative programming, and rule-based systems. CNP is especially suitable for solving problems that have a natural graph-like representation, and/or whose description exhibits nondeterminism and declarativeness. WinSpider is the most current integrated CNP development environment. The paper presents the general picture of how WinSpider can be effectively used for implementing various search strategies. Two major implementation techniques are outlined. In the first approach, the classical search algorithms are essentially simulated in CNP – this type of implementations is referred to as procedural implementations. A generic search CNP solution is described which can be used for modeling many fundamental strategies for uninformed or heuristic search. Then, for a group of search strategies called local search strategies, a very different and interesting approach is shown where the CN simply describes the problem but includes no explicit procedure for implementing the search process at all; instead, dynamic control system options and control states are used in order to enforce the interpreter to "automatically" perform the desired search strategy. This second type of implementation is called non-procedural. Finally, certain complex search strategies such as simulated annealing require a combination of both techniques.***

Key words: ***Search strategy in WinSpider, WinSpider, CNP, Control network programming, Search algorithm, Local search, Hill climbing, Simulated annealing.***

1. Introduction

Control Network Programming (**CNP**) evolved as a programming paradigm by combining features from imperative programming, declarative programming, and problem solving with rule-based systems, at the same time substantially extending their potential [1].

1.1. Control Network Programming

The program in CNP can be visualized as a finite set of graphs referred to as a **control network** (**CN**). The graphs comprising the CN are called subnets; one of them is identified as the main subnet. Each subnet consists of nodes (states) and arrows. A chain of "primitives" is assigned to each arrow. The primitives may be thought of as elementary actions and are technically user-defined procedures. A subnet may call other subnets or itself. Both subnets and primitives can have parameters and variables. The complete program consists of two main components - the CN and the definitions of the primitives.

The program is a possibly nondeterministic description of a problem. "Executing" the CN means traversing the CN starting from the unique initial node of the main subnet and executing the primitives along the way. This process will successfully finish when the interpreter arrives at a system node *FINISH*. The computation/search strategy is an extended version of backtracking; one of the major enhancements is the possibility to backtrack through a subnet.

Copyright © 2011 by Technical University Sofia, branch Plovdiv, Plovdiv, BULGARIA. ISSN 1310 - 8271

CNP allows a convenient combination of procedural and non-procedural features – while the structure of the CN is generally non-procedural (i.e., can be a simple representation of the problem without specifying any algorithm for solving this problem) and nondeterministic, the primitives used in it are procedural. However, procedural solutions can also be easily programmed.

The syntax and semantics of CNP, and more specifically of the CNP programming language *Spider*, have been described in [2, 5, 6]. Representative examples of using CNP for solving various types of problems have been considered in [3]. Programming environments for developing and running CNP applications are available for free download at [15]; the code of all examples from the mentioned publications has also been posted there.

CNP supports powerful means (system options and control states) that give the programmer extensive control over the computation (inference). Using these means the programmer can improve the efficiency of the computation and easily implement various types of heuristic algorithms. These control features and their usage are studied in [1, 4, 7].

1.2 Purpose of this report

Search algorithms play a fundamental role in Artificial Intelligence. They also attract considerable interest in Algorithm Design, Computability Theory, Operations Research, some areas of Mathematics and Engineering, Robotics, Bioinformatics, and other fields.

Numerous examples of using CNP for implementing search strategies are discussed in [4]. The focus is on employing the built-in tools for dynamic control of the computation process for automatic, non-procedural modeling of certain search strategies.

This report studies the possibilities for programming search algorithms in CNP in a more general framework. Two major implementation techniques are outlined. In the first approach, the classical search algorithms are essentially simulated in CNP – we refer to such implementations as **procedural implementations**. We describe a generic search CNP solution which can be used for modeling many fundamental strategies for uninformed or for heuristic search. Then, for a group of search strategies that we call local search strategies, we show a very different approach where the CN simply describes the problem but includes no explicit procedure for implementing the search process at all; instead, dynamic control system options and control states are used in order to enforce the interpreter to "automatically" perform the desired search strategy. We refer to this type of implementations as **declarative (non-procedural) implementations**. Finally, certain search strategies require an approach that combines features of both procedural and non-procedural implementations.

The *Spider* code of all examples described here is available at [15]. Typically, each strategy is applied for two particular problems – the road map problem [4,6,7,11,12] and the 8-puzzle problem (e.g., [11-13]). The specific map used in the examples is the one shown in Fig.7 of [6] and Fig.3 of [7].

2. Procedural Implementation of Search Strategies

This approach can be used for implementing many of the fundamental search strategies, both blind and informed; the strategies we cover are breadth-first, depth-first (leap frogging), uniform-cost, best-first, A*. The strategies use data structures usually called CLOSED and OPEN containing, respectively, the nodes already explored and the nodes at the fringe of the search. Each node may be accompanied by some additional information such as the cost from the initial node to this node, a heuristic evaluation of the cost from this node to a final node, a parent of the node. While CLOSED is theoretically a set, container OPEN is ordered (the order depending on the strategy being implemented).

2.1 The generic search algorithm

In order to come up with a CNP solution, we develop first the pseudo-code of a generic search algorithm that generalizes all mentioned strategies. Then we create the corresponding UML activity diagram and (in an almost trivial manner) convert this diagram into a CN. A similar conversion was used in [3] for solving the Selection Sort example.

```
procedure Generic_Search;
begin
  OPEN← [makeInitEntry(START)];
  CLOSED← [ ];
  while OPEN<>[ ] do
  begin
   popOpen(S);
   if final(S) then return(solution)
   else begin
            pushClosed(S);
            findChildren(S);
            pushOpen;
            sortOpen;
         end; {else}
  end; {while}
  return(failure);
end; {Generic_Search}
```

Fig. 1. *The generic search algorithm*

The generic search algorithm is unique and works for all strategies; the strategy is being chosen through a dialogue with the user. The generic

algorithm is based on the observation that all other search strategies under consideration are actually special cases of the A* algorithm. A similar but narrower universality observation can be found in [12, 13].

```
procedure pushOpen;
begin
for each T=<child,g,h,parent> in the set of the children do
    begin
    case
        CLOSED contains an element <child,g',?,?>:
            begin
            if g < g' then
                begin
                delete <child,g',?,?> in CLOSED;
                add <child,g,h,parent> to OPEN;
                end
            end
        OPEN contains an element <child,g',?,?>:
            if g < g' then
                replace <child,g',?,?> by <child,g,h,parent> in OPEN;
        else:
            add <child,g,h,parent> to OPEN;
end; {case}
    end; {for each}
end; {pushOpen}
```

Fig. 2. *Procedure pushOpen*

The pseudo-code of the generic algorithm is shown in Fig. 1. It uses function *pushOpen* specified in Fig. 2.

We distinguish between a state and a **complete state**. The latter is an ordered quadruple <state,*g*,*h*,*parent*> where *g* and *h* are values called **cost** and **heuristic evaluation**, respectively. The last component, *parent* is used for restoring the solution path after finding a solution. OPEN is a set of complete states. At each step of the algorithm it is re-ordered with respect to a strategy-dependent function, *f* referred to as the **total function**.

Function *makeInitEntry(state)* converts *state* into a corresponding complete state *<state,0,?, nil>*. CLOSED is a set of complete states. Procedure *popOpen(S)* assigns its output parameter *S* of type complete state the value of the first element of OPEN, and removes this element from OPEN. The *solution* is typically a path of states. It can be restored from the final state using the *parent* components in CLOSED. It might also contain additional information such as the cost of the solution. Procedure *findChildren(S)* generates the set of all successors of the complete state *S* – the current state. This procedure is problem-dependent. If *S=<state,g,h,parent>* then, for each successor, *newState* of *state*, the cost of *newState* will be calculated as follows: for A* and uniform-cost search *newState.g := state.g+arc_cost (state,newState)* where *arc_cost(state,newState)* is the cost of the arrow between the two states; for best – first, breadth - first and depth - first search

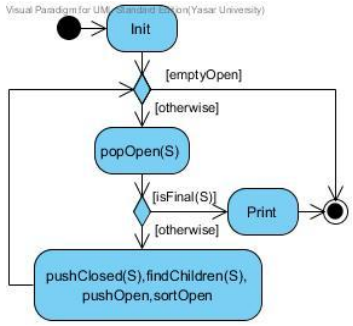

Fig. 3. *The generic search algorithm - activity diagram*

strategies *newState.g:= state.g+1* (i.e., depth). The heuristic evaluation of the child is calculated by a problem-dependent function, *h*. The value *newState.parent* is set to *state*. Procedure *pushOpen* is strategy-independent. Some of the children of *S* are killed; the surviving children are pushed into OPEN. The algorithm of this procedure becomes clear from Fig. 2. Set OPEN is sorted in ascending order according to the values of the following algorithm-dependent total function: for a given complete state *T=<state,g,h,parent>* the value *f(T)* is equal to *g+h* if the strategy is A*, equal to *h* for best-first search, to *g* for the uniform-cost and breadth-first strategies, and to *–g* for depth-first search. We assume here that optimal for both costs and heuristic evaluations means the smallest.

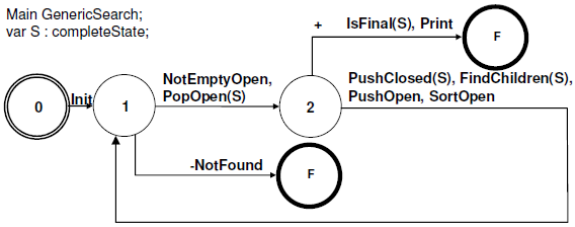

Fig. 4. CN for the generic search algorithm

The activity diagram of the generic search algorithm is shown in Fig. 3. The CN of the CNP implementation is illustrated in Fig. 4.

Note that the CNP implementation of the generic search algorithm discussed above is universal and covers all the search strategies under consideration. The actual strategy is chosen by the user during an initial dialogue (in which the initial and the final states are also specified). In our implementation, the code of all primitives is generic and does not depend on the strategy. The choice of the strategy made by the user is used in two simple supporting functions – one employed in primitive *findChildren* to calculate *g*, and one in primitive *sortOpen* for calculating *f* (see the explanations above).

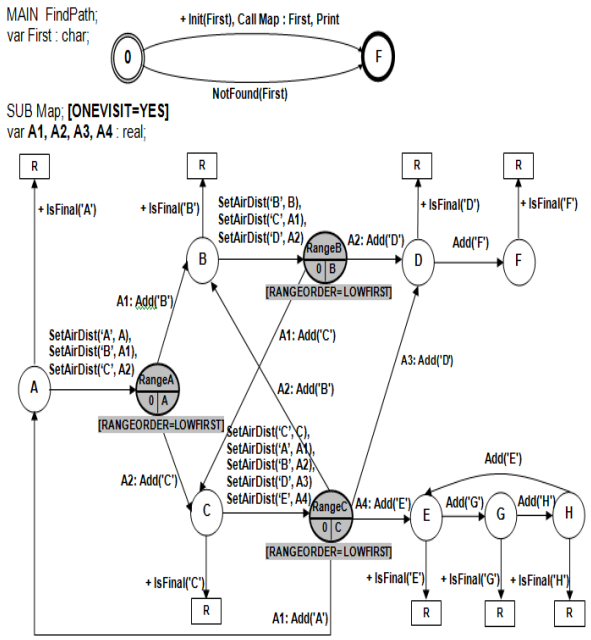

Fig. 5. *Non-recursive hill-climbing*

In addition to being strategy-independent, the CN is also problem-independent - the same for different problems for which a given search strategy is applied (e.g., for the road map and 8-puzzle problems). As far as not only the CN but the complete CNP code is concerned, certain components must necessarily be problem-dependent. In our particular implementation (available at [5]), problem-dependent are a few help functions such as *arc_cost* and *h*. Of course, a few problem-dependent data structures will also be needed such as, for example, the representation of the map in the road map problem.

2.2 Using strategy-dependent solutions

One could also write separate (and probably more efficient) CNP implementations for a specific strategy. Even for such an algorithm the CN will remain unchanged. Simply, some of the primitives used must be adjusted to the particular strategy. Mainly, this will affect primitives *pushOpen* and *sortOpen*.

2.3 Abstraction and expressiveness of CNP procedural implementation

The CNP simulations of the search strategies discussed in Section 2 are simple to apprehend, design and implement but at the same time are rather general and abstract. (Well, one should expect similar characteristics from a CNP solution.) It is worth mentioning that ideas similarly related to abstraction and generality can be found in well-designed object-oriented implementations of the search algorithms (e.g., [10]). These implementations use advanced object-oriented programming concepts such as abstract classes and interfaces. We would like to note that a CNP solution in a CNP environment with an object-oriented underlying language (environments such as *WinSpider* or *Spider#* - the latter being currently developed) could combine such typical advanced OOP possibilities with the possibilities offered by CNP itself.

The CNP solutions in Section 2, however, do not show the really great potential of CNP. The reason is the fact that the above implementations actually emulate procedural algorithms and, consequently, *do not involve any non-determinism.*

3. Non-Procedural Implementation of Local Search Strategies

In order to be able to create non-procedural search implementations we must make use of the built-in in CNP search mechanism which is an extended version of backtracking [1, 2, 5-7]. Then we will be able to use a descriptive-type CN, and it is the built-in interpreter's responsibility to "compute" the CN by finding a successful path from the initial to a final node.

Trivially, if the search strategy we want to model is backtracking (or we don't care what search strategy will be used) then there is no need to write a corresponding search procedure at all – we simply declaratively describe the problem and leave the inference to the interpreter. Eventually, the CNP programmer might want to use some of the static search control tools [6] for a better efficiency, to solve some problems related to non-termination, to specify other requirements such as the maximum number of the solutions required or the maximal length of the solution paths.

There are numerous other search strategies – both heuristic and improved uninformed ones - that have evolved from backtracking. Non-procedural implementations of some of them were described in [4] with complete codes shown at [15]: optimal search with cutting off insipid paths (branch-and-bound), hill climbing, irrevocable hill climbing, nearest neighbor search, version of beam search, stochastic hill climbing, first-choice hill climbing. In all these implementations we have made use of the so called tools for dynamic control of the computation – dynamic control system options and control states [4, 7]. Applying such tools the CN programmer can easily model different modes of choice of which arrow emanating from the current node to attempt first (such as the seemingly best first, or randomly, or within a range of evaluations, or a restricted number of arrows, etc.).

The programmer can also modify other parameters of the backtracking (e.g., forbid backtracking). Such modifications can be even performed dynamically (using variables whose values can be changed during computation).

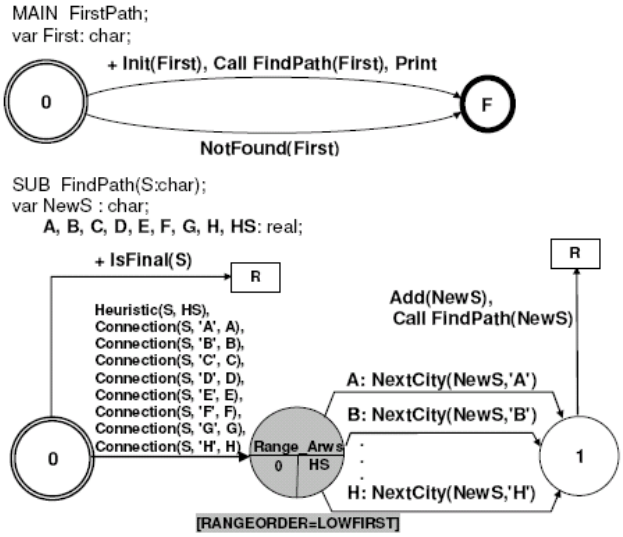

Fig. 6. *Recursive hill-climbing*

Basically, a programmer can do anything they want with the arrows of the current state. The control may also backtrack to a neighboring state which then becomes current. We refer to this group of search strategies as "local search strategies". This concept is almost identical with the notion of "local search" as usually understood in the literature [8, 9, 12, 14]. In the referenced sources, however, the emphasis, setting and viewpoint are somehow different. These methods are being applied for optimization problems (satisfiability problems or pure optimization problems [14]). There is no reason why a heuristic function should be treated differently from an objective function though, and a space of solutions with solutions being states is actually no different from a state space. For example, the usage of the Best Improvement systematic heuristic [14, para. 2.2.1] yields the classical hill-climbing strategy.

Local search in the literature refers to selecting a legal neighbor of the current state [e.g., 14]. In CNP the control moves from the current state to a neighboring state along an emanating arrow. Therefore, CNP is 'intrinsically local' which allows an easy and natural non-procedural implementation of local search strategies making use of the rich set of means for computation control. Note, that our local search is usually revocable, in contrast to the algorithms from [14] and elsewhere.

For more information on implementations of local search strategies the reader is referred to [4, 7]. Only one example is shown here in Fig. 5 – one of the three discussed implementations of the hill climbing algorithm for a given road map that includes cities A through H.

When the state space is large, it is preferable to apply recursive CNP solutions. A recursive equivalent of the hill-climbing strategy for the road map problem is shown in Fig. 6. More details can be found in [4]. Note that the recursive solution remains non-procedural – hidden for the programmer backtracking is possible back through subnet calls.

4. Mixed Implementations

Not all local search algorithms, however, are suited for fully non-procedural CNP implementation. Some more complex strategies include elements that need procedural implementation. The resulting CN solution should be called a mixed implementation.

An example (from [4]) is shown in Fig. 7. It represents a solution to the Traveling Salesperson Problem (for cities A, B, C and D) using the Simulated annealing heuristic strategy. *MonteCarloStep* subnet defines declaratively a 'chunk' of the state space. The outer loop typical for the simulated annealing strategy, is modeled by the main subnet.

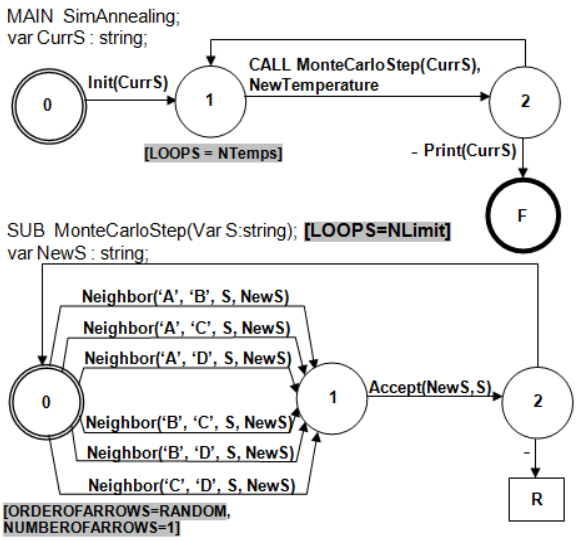

Fig. 7. *Simulated annealing*

It is not surprising that the Simulated annealing algorithm requires a hybrid implementation. As noticed in [14, para. 2,3,2], Simulated annealing is a metaheuristic based on the Metropolis heuristic. While the Metropolis heuristic can be readily implemented non-procedurally (heuristics drive the search towards local optima), the metaheuristic needs to be simulated procedurally (metaheuristics collect global information and aim primarily at escaping local

optima driving the search towards global optimality).

5. Conclusion

The report identifies, classifies and illustrates the ways in which search problems may be approached by a CNP programmer. Many local search strategies based on or derived from backtracking and hill-climbing can be implemented non-procedurally using the supported in CNP tools for computation control. Non-local algorithms can be modeled using directly the presented generic procedural CNP solution or at least using the underlying pattern. Finally, more complex algorithms may require a mixed approach comprising elements of both types – procedural and non-procedural – at different levels.

References

1. Golemanova E., K. Kratchanov, T. Golemanov Spider vs. Prolog: Computation Control, In: *10th Int. Conf. on Computer Systems and Technologies (CompSysTech 2009)*, Rousse, Bulgaria, 2009, pp. II.10-1-II.10-6, Also: *ACM International Conference Proceeding Series*, Vol. 433.

2. Kratchanov K., E. Golemanova, T.Golemanov Control Network Programs and Their Execution, In: *8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009)*, Cambridge, UK, 2009, pp. 417-422.

3. Kratchanov K., E. Golemanova, T.Golemanov Control Network Programming Illustrated: Solving Problems With Inherent Graph-Like Structure, In: *7th IEEE/ACIS Conf. on Computer and Information Science (ICIS 2008)*, Portland, OR, USA, 2008, pp. 453-459.

4. Kratchanov K., E. Golemanova, T.Golemanov, T,Ercan. Non-Procedural Implementation of Local Heuristic Search in Control Network Programming, In: *Knowledge-Based and Intelligent Information and Engineering Systems, Proc. 14th Intl Conf. (KES 2010), Cardiff, UK, Sep 2010, Part II, Lecture Notes in Artificial Intelligence),* v.6277, Springer, 2010, 263-272.

5. Kratchanov K., T. Golemanov, E. Golemanova Control Network Programming, In: *6th IEEE/ACIS Conf. on Computer and Information Science (ICIS 2007),* Melbourne, Australia, 2007, pp. 1012-1018.

6. Kratchanov K., T. Golemanov, E. Golemanova Control Network Programs: Static Search Control with System Options, *In: 8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009)*, Cambridge, UK, 2009, pp. 423-428.

7. Kratchanov K., T. Golemanov, E. Golemanova, T. Ercan Control Network Programming with SPIDER: Dynamic Search Control, In: *Knowledge-Based and Intelligent Information and Engineering Systems, Proc. 14th Intl Conf. (KES 2010), Cardiff, UK, Sep 2010, Part II, Lecture Notes in Artificial Intelligence*), v.6277, Springer, 2010, 253-262.

8. Levitin A. *Introduction to the Design and Analysis of Algorithms*, 2nd ed., Addison-Wesley, Boston, 2007.

9. Luger G. Artificial Intelligence: Structures and Strategies for Complex Problem Solving, 6th ed. Addison-Wesley, Boston, 2009.

10. Luger G., W. Stubblefield *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java*, Addison-Wesley, Boston, 2008.

11. Pearl J. Heuristics. Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.

12. Russell S., P. Norvig *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, Upper Saddle River, NJ, 2010.

13. Thornton C., B. du Boulay *Artificial Intelligence. Strategies, Applications, and Models Through Search*, 2nd ed. Amacom, New York, 1998.

14. Van Hentenryck P., L. Michel *Constraint-Based Local Search* MIT Press, Cambridge, MA, 2005

15 www.controlnetworkprogramming.com.

[1]Department of Software Engineering
Yasar University
Universite Cad. No.35-37
35100 Bornova/Izmir
Turkey
E-mail: kostadin.kratchanov@yasar.edu.tr
E-mail: yasemin.gokcen@stu..yasar.edu.tr

[2]Department of Computing
Rousse University
8 Studentska St.
8000 Rousse
Bulgaria
E-mail: EGolemanova@ecs.ru.acad.bg
E-mail: TGolemanov@ecs.ru.acad.bg