# Control Network Programming: Static Search Control with System Options

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

[1] Department of Computer Engineering
Yasar University, Izmir, TURKEY
kostadin.kratchanov@yasar.edu.tr

[2] Department of Computing
Rousse University, Rousse, BULGARIA
(TGolemanov, EGolemanova)@ecs.ru.acad.bg

*Abstract:* Control Network Programming (CNP) is a programming paradigm that is especially convenient for representing problems with a natural graph-like description. This description is often of nondeterministic nature. The report continues the discussion from [1] on how a CN program is interpreted and executed. The basic algorithm of the interpreter is presented. It is followed by a discussion of the so called system options – these are powerful means for user control of the execution process.

*Key-Words:* Control network programming, CNP, control networks, system options, interpreter, search control, computation control, programming languages, AI programming, nondeterministic programming, non-procedural programming, Spider, search.

## 1 Introduction

CNP [1-3] started as an attempt to enhance the possibilities and flexibility of controlling the inference process in rule-based systems by equipping the rule base with a structure, i.e., arranging the rules into a "control network". It developed into a universal programming paradigm that is especially effective for solving problems with natural graph-like representation [3]. The problem in hand may be nondeterministic and its specification purely descriptive.

The structure and syntax of a CN program were introduced in [2], and typical representative problems considered in [3]. CN programs and their behavior were more formally defined in [1], and the basics of their execution introduced.

The discussion in [1] focuses on the main concept of CN program execution only. There are additional, more advanced features that were left completely untouched. First of all, these are the powerful means for controlling the search provided by the so called control nodes and system options. Specifying a group of these features – the system options for static control - and illustrating their usage comprise the main subject of this report. It is assumed that the report will be read in conjunction with [1-3]. For consistency, the CNP development environment Spider, respectively the Spider CN programming language, are used.

## 2 Algorithm of the interpreter

The interpretation of a CN program was discussed in detail in [1].

The actual formal algorithm of the CN program interpreter is specified in Figures 1 and 2 with its UML activity diagrams. "Process CN" is the main activity. It invokes activity "Process Node". A short explanation of some of the notations and data structures used follows.

The set *out(v)* includes all arrows outgoing from vertex *v*. *RET_stack* is a stack where states immediately following subnet calls are stored while *remains* is the set of the arrows outgoing from the current state and not attempted yet. *arr_done* is a stack of primitives from the current arrow label that have been executed already. *arr_to_do* is the remaining sequence of primitives associated with the current arrow *arr*. Operation *push(a, stack)* pushes element *a* onto stack *stack*. Operation *pull(a, stack)* stores the top of the stack into *a*, and removes the top of the stack. Operation *dequeue(a, queue)* stores the element at the front of the queue *queue* into *a*, and removes this element from the front of the queue.

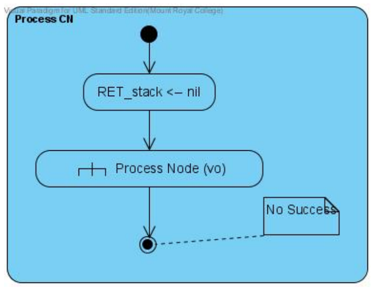
Visual Paradigm for UML Standard Edition(Mount Royal College)
Process CN
RET_stack <-- nil
Process Node (vo)
No Success

**Figure 1 Activity diagram of "Process CN"**

Returning backwards into a subnet poses a specific problem. To develop a solution around this difficulty, we need to impose on the CN the following two constrains: An invocation *CALL(N':v')* may only be the last component in an arrow's label, and the subnets' state

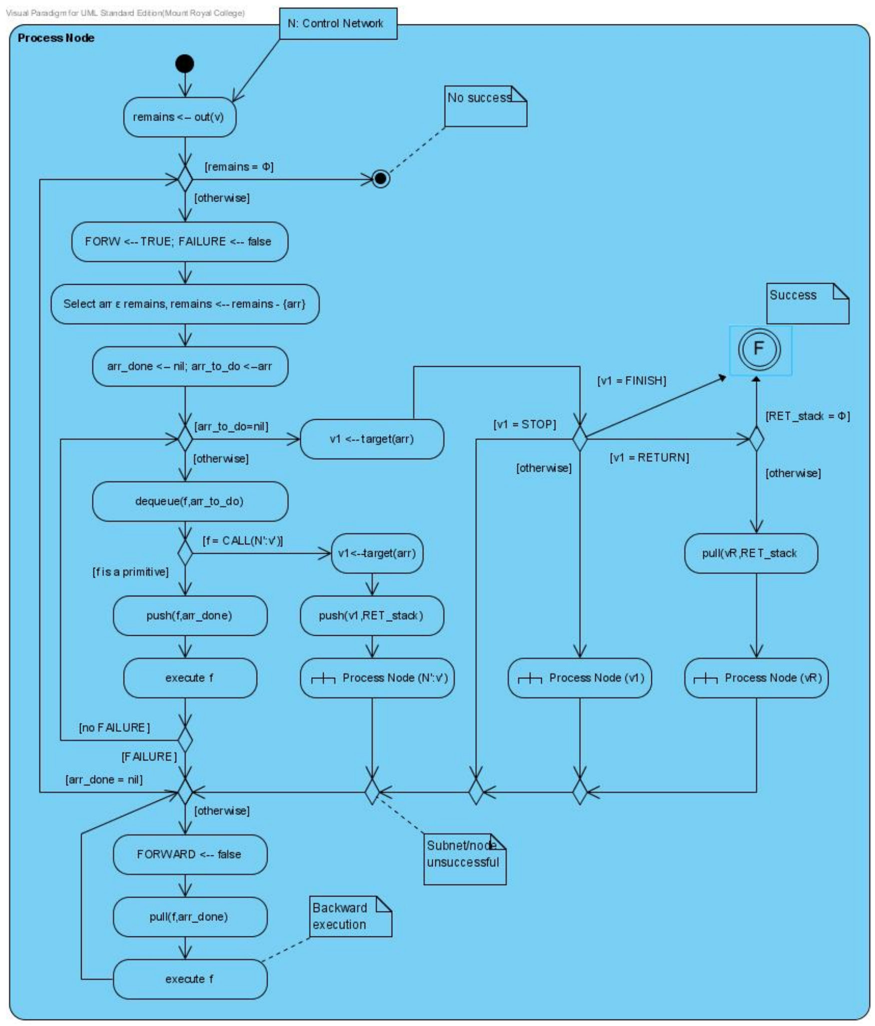

**Figure 2 Activity diagram of "Process node"**

sets are non-intersecting (i.e., all subnets have unique states).

These restrictions do not decrease the generality of our considerations. Indeed, if the original CN contains an arrow in which an invocation is not the last component, then such an arrow can always be replaced by an equivalent sequence of arrows. For instance, the arrow $v' \xrightarrow{p1, CALL1, p2, CALL2, p3} v''$ can be replaced by

$$v' \xrightarrow{p1, CALL\ 1} v1 \xrightarrow{p2, CALL\ 2} v2 \xrightarrow{p3} v''.$$

In regard to the state names, we can always use the coproduct (disjoint union) of the original state sets..

As a matter of fact, the above two restrictions do not affect programming in Spider. As explained in [1], the original Spider program is compiled into an intermediate program in the underlying programming language that embodies information on both the original CN program and the algorithm of its interpretation. This "compiler" also acts as a "pre-processor" automatically converting the original CN into a CN that satisfies the above restrictions.

Due to the first restriction, it is possible to store onto stack *RET_stack* the state immediately following a subnet call (see Figure 2). The second restriction is needed because the subnets are not modeled by procedures calling each other as that won't allow for returning backwards into a subnet – instead, all subnets are "projected" into a single plane.

Completion of a CN or a subnet means that all possible paths have been attempted. Therefore, completing the CN corresponds to an unsuccessful search. This is the meaning of the normal exit nodes in the activity diagrams in Figures 1 and 2. There is another possible way to exit a CN or a subnet – when a *FINISH* state has been reached by the search process. This is the situation when the search was successful. In technical terms, this situation is not a normal exit from the CN. Such an event can be better described using terms like HALT, ESCAPE, ABORT or BREAK. The issue of multi-exit control flow has attracted lots of attention in certain periods of the programming languages development (see, e.g., [4, Chapter 11; 5, Chapter 8]). As argued there, a typical situation when a break will be advantageous is a search loop. In CNP, all the computation is essentially a search. Therefore, it is understandable that we found it natural and effective to abort the computation completely when the control reaches a *FINISH* state. UML does not provide means for depicting multi-exit control. Therefore, we have used the traditional activity end symbol to depict an unsuccessful completion of the algorithm and adopted a special new symbol (Ⓕ) to denote the successful halt of the process.

## 3 On the extended backtracking

The computation process is essentially a process of searching the CN with the purpose of finding a solution path from the default initial node of the main subnet to a *FINISH* state. The method used in Spider extends the well-known backtracking algorithm, adding also some unusual features.

Firstly, our setting is not traversal in a graph but rather traversal in a CN which is a set of graphs. The control may have to enter a subnet backwards through a

*RETURN* node, and in some cases to pass backwards the whole subnet (see [1] for examples). Entering a subnet backwards is not a trivial problem. A subnet plays the role of a procedure/function. Normally, when a procedure/function in a programming language is completed, the memory allocated for it is freed and all related information is lost. Therefore, we had to set aside the natural idea to implement a subnet through a procedure, and find a different approach.

Secondly, we decided against storing (in a return stack) of all the information related to a state in the CN. This information must normally include not only the position in the CN itself, but also all the data. Therefore, we have introduced forward and backward execution of a primitive – the latter one conducted during backward execution in order to restore the state of the data. This approach can be seen in the code of the *IncPr* primitive in [1]. Theoretically, in general, it is not always possible to find the inverse behavior of a primitive because the inverse of a function is in general a relation. In practice, however, we came to the recognition of the following intuitive rule: a programmer never uses a primitive that has no inverse action in a place where such an inverse action would be actually necessary. In our experience with CNP we have never observed an exclusion from this rule.

## 4 Control features in Spider

Solving certain problems requires that the standard interpretation algorithm be modified in one way or another. **System options** supported in Spider provide possible ways of doing so and are a powerful means for controlling the search in the CN.

Some of the system options are used in conjunction with the so called **control states** which are a second group of search control features. These are special types of nodes in the CN: *ORDER*, *SELECT* and *RANGE*.

Spider offers ten system options. According to their main purpose, the system options can be classified in three groups:

- For setting the solution scope – system option *SOLUTIONS.*
- For static control of search parameters – system options *BACKTRACKING, ONEVISIT, LOOPS* and *RECURSION*.
- For dynamic search control (controlling the order of selecting the outgoing arrows of the active state) – system options *ARROWCOST, MAXPATHCOST*, *NUMBEROFARROWS, RANGEORDER,* and *PROXIMITY.*

This presentation focuses on the system options of the first two groups only. These system options are summarized in the following table:

| Option | Default value | Other values |
|---|---|---|
| **SOLUTIONS** | 1 | ALL, ASK, unsigned integer |
| **BACKTRACKING** | YES | NO |
| **ONEVISIT** | NO | YES |
| **LOOPS** | ANY | Unsigned integer |
| **RECURSION** | ANY | Unsigned integer |

The static control options allow for the elimination of certain algorithmic issues (e.g., avoiding unwanted loops) or for improving the search efficiency.

The dynamic control options are used, in conjunction with control states, for implementation of heuristics. Dynamic control involves changing the values of system options during the computation process.

At each point of the execution, every one of the ten system options has a value (stored, together with certain additional system information, in a variable – record called *SpiderOptions*). The values can change. At a given point, the set of the ten system option values determines the exact behavior of the interpreter.

The possible values of the system options under consideration are listed in the table. To set an option's value, the following syntax is used: *[option name = value]*. The value of a system option is set immediately before the segment of the CN which the setting is to affect. Such a segment (the **option scope**) may be the entire CN, a subnet, a state (the scope is the set of outgoing arrows for that state), or an arrow. For some options certain scopes make no sense and cannot be used. A setting with its scope being the entire CN is declared at the very beginning of the CN (right after "(*&N" [2]). A setting with any of the other scopes is declared immediately after the name of the corresponding fragment (e.g., "*SUB Map; [LOOPS=0]*"). In the case an option setting has not been specified for a given fragment, the option's value from the segment immediately including the given segment will be valid. Each option also has a default value (see the table) – that is the value of the option if no explicit setting is specified. The "basic" interpreter algorithm of

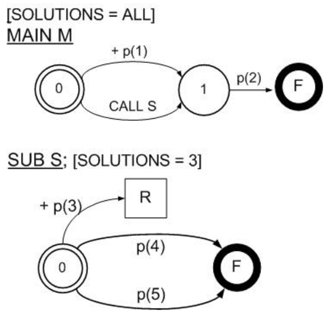

**Figure 3 An example: CN**

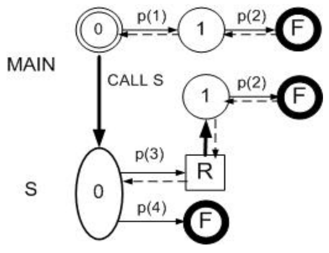

**Figure 4 An example: execution**

Figures 1 and 2 corresponds actually to the default option values.

## 5 System option SOLUTIONS

The value of the option determines the number of solutions that the interpreter will seek. Setting *[SOLUTIONS = 1]* instructs the system to halt after finding the first solution. If the value of the option is *ALL* the interpreter interprets state *FINISH* as "no success" and continues the computation, until the CN is completely traversed. Setting *ASK* will make the program print a message asking the user whether to continue with the search. The option may be used for the entire CN, a subnet (only if the subnet contains a *FINISH* state), a state (only if the state contains an outgoing arrow with target state *FINISH*).

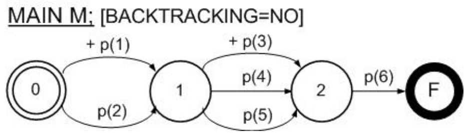

**Figure 5 Another example: CN**

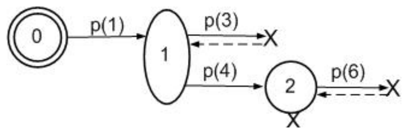

**Figure 6 Another example: execution**

In Figure 3, *[SOLUTIONS = ALL]* is specified for the entire CN, and then redefined as *[SOLUTIONS = 3]* for subnet *S*. The execution is illustrated in Figure 4. We assume that all the primitives will be executed successfully. After executing primitives *p(1)* and *p(2)* the control reaches state *FINISH* of *MAIN*. The first solution has been found but the computation does not halt, *p(1)* and *p(2)* are executed backwards, and the control reaches state *0* again. The second arrow outgoing from state *0* is attempted now by executing the subnet call *CALL S*. The value of *SOLUTIONS* for subnet *S* is 3. The first outgoing arrow is taken, primitive *p(3)* is executed and state *RETURN* is entered. As subnet *S* was successfully completed, the control moves to state *1* in *MAIN*. The value of *SOLUTIONS* changes back to *ALL*. After executing primitive *p(2) FINISH* is reached once again, which means the second solution was found. The computation continues, and the control goes back to *1* through *p(2)*. Subnet *S* is entered backwards through *RETURN*. The value of *SOLUTIONS* restores to *3*. Computation continues as illustrated in Figure 4. Three solutions will be found.

Note that if state *FINISH* in subnet *S* is replaced by *RETURN* then the only *FINISH* state will be in *MAIN* where the value of option *SOLUTIONS* is *ALL*, and therefore all four potential solutions will be found.

## 6 System option BACKTRACKING

This system option is used when the programmer wants to "switch off" the backtracking. In that case, if all the outgoing arrows of a state have been attempted, the control does not move back through the arrow along which the state was entered - instead, the computation halts unsuccessfully. The scope of this option can be the entire CN, a subnet, or a state.

The execution of the CN from Figure 5 is shown in Figure 6. Assume that primitive *p(1)* is successfully executed but then the execution of *p(3)* is unsuccessful. The control returns to state *1* after which the arrow with primitive *p(4)* is attempted. The single outgoing arrow from state *2* is tried. Assume that primitive *p(6)* is not successful. The control returns to state *2*. There are no more arrows outgoing from state *2*. Because the value of option *BACKTRACKING* is *NO*, return from state *2* one step back to state *1* is not allowed. Therefore, the computation stops. If *[BACKTARCKING = NO]* was not used then many more arrows from states *1* and *0* will be attempted; however that would not change the final result (provided the additionally executed primitives did not change the result of *p(6)*). The usage of the system option *BACKTRACKING* improved the efficiency of the computation!

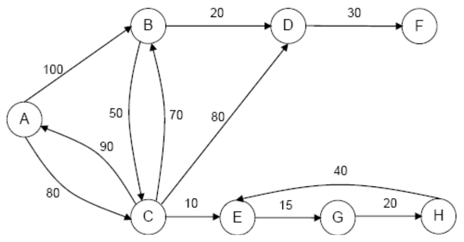

**Figure 7 The map traversal problem**

For the same reason, switching off backtracking was used in the CNP solution to the animal identification problem in Figure 2 of [3]. *BACKTRACKING=NO* can be also employed for realization of irrevocable hill-climbing [7, Ch.11] in conjunction with the usage of control states *ORDER* and *RANGE* [8].

## 7 System option ONEVISIT

This option defines whether or not a specific node can be entered more than once (within a particular recursive level). This setting will assist in avoiding infinite loops as well as unnecessarily traversing fruitless branches of the CN. The possible scopes for this option are the same as for option *BACKTRACKING*.

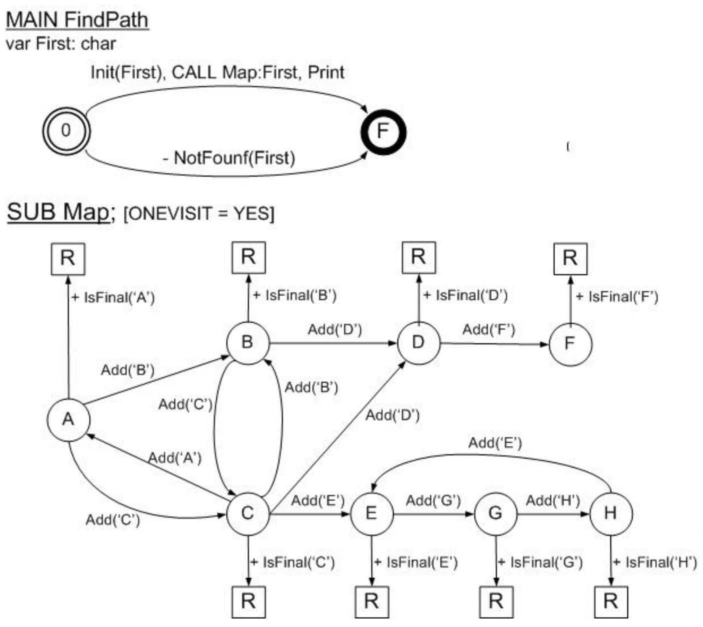

**Figure 8 CN for the map traversing problem (one solution)**

The example we will use is based on a modified version of a problem from [6, p.64]. We will refer to it as the **map traversal problem**. We are required to find an acyclic path from an initial city to a target city using a highway map such as the one shown in Figure 7.

Our approach will be to convert the map into a CN, and leave the built-in Spider search algorithm find a solution. The CN will consist of two subnets (Figure 8). Subnet *Map* represents the map a state corresponding to each city. The programmer only has to take care of forming the solution path – this will be achieved by using a stack where only the city names are stored. Primitive *Add(city)* simply pushes its parameter onto the stack. A local variable, *First* of the main subnet is used to store the name of the initial city. The value of the global variable *Final* is the name of the target city. The values of both *First* and *Final* are initialized by primitive *Init*. Other solutions are also possible.

The value of the system option *ONEVISIT* is set to *YES* for the subnet *Map* only. There are two reasons for doing that. Firstly, this eliminates the creation of loops in a path (e.g., *A*→*B*→*C*→*A*). Secondly, the usage of the option will cut out branches that are not promising. For instance, if *First* = *A* and *Final* = *E*, the control will visit the nodes of *Map* in the following order: *A*→*B*→*D*→*F* →(back to) *D*→(back to) *B*→*C*→*A*(second visit - loop) →(back to) *C*→*B*(second visit - loop)→(back to) *C*→*D*(second visit – not promising)→(back to) *C*→*E* (success). Indeed, state *D* is not promising, the control has been there already and no solution has been found after exploring the CN from this state.

## 8 System option LOOPS

A loop is a repeating entering into a given node of the current path. Value *0* for option *LOOPS* does not allow entering a node for a second time, value *1* will not allow entering a node for a third time, and so on. Value *ANY* means no restrictions exist. Clearly, *LOOPS=0* forbids the existence of recurring nodes in the current path, and therefore in the solution path as well.

System option *LOOPS* is similar to *ONEVISIT* - the difference is that *LOOPS* considers how many times a node exist in the current path while *ONEVISIT* counts how many times a node is entered in the process of computation. For the example of Figure 8, the transition *C*→*D* is not allowed because *D* was already entered earlier, although *D* was unsuccessful and was not left in the current path. Therefore, if we replace *[ONEVISIT=YES]* with *[LOOPS=0]*, loops will be eliminated but the second visit to node *D* will not be barred and therefore the computation will be less efficient. Also, *LOOPS=n* for a positive integer *n* will allow loops with a length less than a certain number which may be needed in some situations, e.g. when we allow some repetition in words but don't want to allow infinite words.

Let us consider an example for the usage of option *LOOPS*. Assume that we would like to find all possible paths in the map traversal problem. The CN of the solution is given in Figure 9. In comparison with the

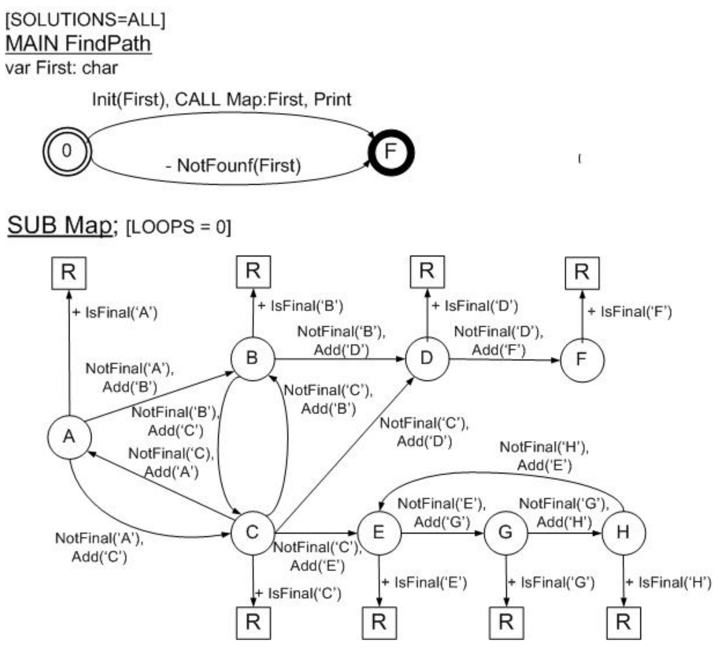

**Figure 9 CN for map traversal problem (all solutions)**

single path solution of Figure 8, *[SOLUTIONS=ALL]* is included before the main subnet and its scope is the whole CN, and *[ONEVISIT=YES]* is replaced by *[LOOPS=0]*. These two changes will make the solution of Figure 8 work. We have made one more adjustment, adding primitives *NotFinal* before each *Add* primitive.

This will improve the efficiency of the computation barring any exploration after the final city (e.g., *E*).

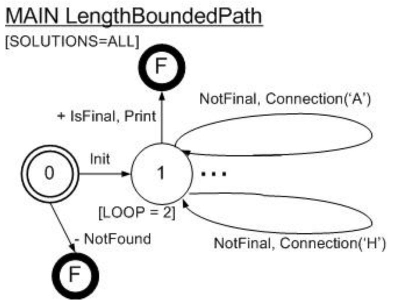

**Figure 10 CN for the map traversal problem (all solutions of limited length, iterative solution**

Another modification of the map traversal problem is solved in Figure 10: find all paths of maximal length 2. This will be achieved employing *[LOOPS=2]*. A fundamentally different approach is applied here. The map is not embodied into the CN, but is given as a global variable - matrix, *Map* showing the distances between the cities. *First* and *Final* are also global variables, as is the current city, *City*. Primitive *Connection(NextCity)* checks in the matrix *Map* if a connection exists from *City* to *NextCity* and if *NextCity* is already a component of the current path – if it is then the transition is not possible. (We can not eliminate the loops using *LOOPS=0* because a loop in the map and a loop in the CN are completely different concepts.) In this solution, *[LOOPS = 2]* has action scope state *1* of the CN. This means that the current path in the computation cannot include state *1* more than three times. For our example the only solution path found will be *A→C→E*; the other potential path *A→B→C→E* will be ignored as its length exceeds 2.

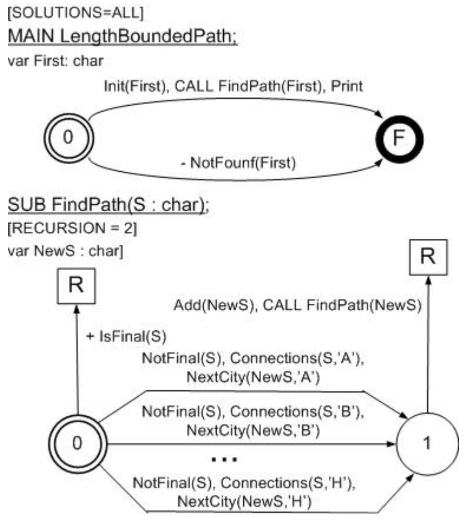
[SOLUTIONS=ALL]
MAIN LengthBoundedPath;
var First: char
Init(First), CALL FindPath(First), Print
0
F
- NotFounf(First)
SUB FindPath(S : char);
[RECURSION = 2]
var NewS : char]
R
R
Add(NewS), CALL FindPath(NewS)
+ IsFinal(S)
NotFinal(S), Connections(S,'A'), NextCity(NewS,'A')
NotFinal(S), Connections(S,'B'), NextCity(NewS,'B')
0
...
1
NotFinal(S), Connections(S,'H'), NextCity(NewS,'H')

**Figure 12 CN for the map traversal problem (all solutions of limited length, recursive solution)**

## 9 System option RECURSION

This system option restricts the maximum number (depth) of subnet calls during computation. Possible scopes are the entire CN or a given subnet. Value *n* of *RECURSION* means that a subnet cannot be entered more than *n+1* times.

As an illustration, Figure 12 presents a recursive solution to the problem from the previous section. Subnet *FindPath* is directly recursive. It has a formal parameter *S* – the current city. Variable *NewS* stores the name of the new current city whose value is formed by primitive *NextCity*. Primitive *Add* modifies the stack for the current path. All solutions of length not exceeding 3 nodes will be found.

## 10 Conclusion

The "basic" CNP interpretation algorithm was presented, and then extended with a description of the effect that the "static" system options have on the search in the CN. The usage of these options was illustrated using practical examples. Another group of system options which control the selection of an outgoing arrow to attempt will be described elsewhere. In conjunction with control states, these system options allow the realization of various heuristic strategies [8,9].

*References:*

[1] K.Kratchanov, E.Golemanova, T.Golemanov: Control Network Programs and Their Execution. This conference.

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

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

[4] A.Fisher, F.Grodzinsky: *The Anatomy of Programming Languages*. Prentice-Hall, 1993.

[5] D.Watt: *Programming Language Concepts and* Paradigms. Prentice-Hall, 1990.

[6] P.Winston: Artificial Intelligence, 3rd ed. Addison Wesley, 1992.

[7] R.Shinghal: *Formal Concepts in Artificial Intelligence*, Chapman & Hall, 1992.

[8] E.Golemanova, T.Golemanov, K.Kratchanov: "Built-in Features of the SPIDER Language for Implementing Heuristic Algorithms", In: *Proc. CompSysTech 2000, Sofia, June 2000*, II.9-1 – II.9-5 (in Bulgarian). Also published by ACM Press, 2091-2095.

[9] K.Kratchanov, T.Golemanov, E.Golemanova, I.Stanev: "Control Network Programming in Spider: Built-In Search Control Tools". In: *New Trends in Artificial Intelligence and Neural Networks* (T.Cificibasi, M.Karaman, V.Atalay – eds.) EMO Scientific Books, Ankara, 1997, 105-109.