Journal of Information Technology Education: Innovations in Practice **Volume 11, 2012**

A Functional Programming Approach to AI Search Algorithms

János Pánovics
Department of Information Technology,
University of Debrecen, Debrecen, Hungary

panovics.janos@inf.unideb.hu

Executive Summary

The theory and practice of search algorithms related to state-space represented problems form the major part of the introductory course of Artificial Intelligence at most of the universities and colleges offering a degree in the area of computer science. Students usually meet these algorithms only in some imperative or object-oriented language (e.g., Java or C#) during the seminars. In this paper, we introduce a new approach for presenting these algorithms to the students, which is programming them in a functional style using the F# programming language.

A couple of years ago, we created a Java class hierarchy for use in our Artificial Intelligence seminars. This well-organized set of classes helps students better understand the operation of the various search algorithms. Since some parts of these algorithms can be more conveniently implemented using a functional approach, we present here the F# implementation of the same class hierarchy. F# proved to be a good choice of programming language because of its multi-paradigm nature. This way, the classes themselves were easy to adopt, and the instructors of the seminars may decide how much of the code they want to rewrite in a functional manner. Functional programming can provide tremendous benefit during the implementation of methods containing logical formulae in their bodies, such as the precondition of an operator. In summary, the power of F# lies not in the fact that it is a functional programming language, but that the developer can select the programming paradigm they want to use in different parts of the program.

In the future, we would like to create a purely functional implementation of the main search algorithms with as much reusable code as possible.

Keywords: artificial intelligence, search algorithms, functional programming, F#, class hierarchy.

Introduction

Until recently, programming was about using pure, single-paradigm techniques. However, nowadays programming languages tend to converge to one another, i.e., functional features are appearing in imperative languages and vice versa. There are a couple of domains of computation in which solutions to problems can be expressed in a more succinct way if we use functional programming compared to using purely imperative or object-oriented tools. We believe that artificial intelligence and search algorithms, in particular, are such domains because some substantial parts of these algorithms (like, for example,

Editor: Keith Willoughby

checking operator preconditions) are essentially functional. There have been numerous publications in this field (e.g., King & Launchbury, 1995).

Teaching search algorithms to our students is a great pedagogical challenge. At our university, they first meet artificial intelligence in the frame of the course *Introduction to Artificial Intelligence*, which is one of the core subjects of our three main undergraduate programs, Software Information Technology, Business Information Technology, and Engineering Information Technology. In the lectures, the pseudocode of these algorithms is presented, together with some examples, but this is not always enough for students to understand what is going on behind the scenes. In the seminars, the instructors show the same algorithms written in a high-level programming language, which used to be Pascal and C, but nowadays we use Java and C#. However, the high-level language code is merely another representation of the pseudocode, so students with little programming background do not find it useful in understanding the operation of the algorithms. The idea is that we should try presenting the search algorithms also by using a very different approach, namely functional programming. A functional program can hide the unimportant steps of searching and focuses only on the problem itself. It may be useful even if students do not have any former knowledge of functional programming, because a functional program is just another way for describing a problem, and not for solving it (although the problem description usually incorporates also at least parts of the solution).

When object-oriented languages were introduced to the curricula of most of our undergraduate programs, we created a Java (and later C#) class hierarchy, which has been used since then in our Artificial Intelligence seminars. Since F# is a multi-paradigm language, i.e., it combines imperative (object-oriented) and declarative (functional) programming techniques, it can be conveniently used to convert our existing Java and C# implementations of AI search algorithms into a partly functional implementation. We hope that this modified implementation of the class hierarchy will help our students better understand the concepts behind the scenes.

Imperative versus Functional Programming

Let's first have a look at the major differences between imperative (procedural) and functional programming paradigms, the latter of which being actually a special case of declarative (non-procedural) programming.

- With an imperative approach, the programmer writes step by step *how* a particular problem can be solved. In contrast, using a functional approach, the programmer only declares *what* the problem is by decomposing it to simple *function* calls.
- The *state* of an imperative program is an important factor, whereas a purely functional program does not have states, because it uses only *immutable* data.
- Because of stateless programming, the execution of a purely functional program does not have *side effects*. This also implies that the order in which the expressions are evaluated is not important. The program will yield the same output on the same input in any evaluation order. This is called *referential transparency*.
- The main building blocks of an imperative program are *statements*, while a functional program consists almost exclusively of *expressions*. Some basic control structures (such as conditionals or loops), which are statements in an imperative language, may also appear in functional programs but only as expressions.
- Unlike in imperative programming, *recursion* plays a significant role in functional programming.

Benefits of Functional Programming

Ever since the first programming language was invented, developers have been using mostly imperative languages. This is because functional programming style requires a special, sometimes mind-bending way of thinking about things. There are, however, a lot of benefits to the functional way. In most cases, it requires much less and clearer code to achieve the same result than imperative programming because of language constructs of a higher abstraction level. Less code also means less chance of errors, less testing, and, due to this, more productivity. Functional programs are less error-prone, can be more easily parallelized, and they can be developed in a shorter time. Because of these advantages, functional programming is becoming more and more popular nowadays; even the software industry is looking for more and more programmers with expertise in functional programming. Also, today it usually occurs in the curricula of graduate (and sometimes also undergraduate) programs in higher education.

Functional Programming in Teaching Artificial Intelligence

Besides coding in an object-oriented language, we propose using also the functional approach for programming the solutions to state-space represented problems that students meet during the courses for the following reasons:

- These are complex problems. We do not teach programming in the frame of this course anymore; instead, we teach how the previously learned programming knowledge can be combined with the theory of search algorithms. The more complex a problem is, the more elegantly it can be implemented using functional programming.
- Some parts of the AI search algorithms are functional by their very nature. The source code of these parts simply looks better in a functional language.
- It is worth implementing a couple of problems and search algorithms with both paradigms so that students can see the difference between them. Later they can decide which approach to use in their homework or during a test.
- Functional programming is an exciting challenge for the students, and challenge can be a great motivating force. They prefer dealing with challenging problems even if those problems are difficult or abstract.

As a proof of the succinctness of a functional program solving an AI problem, we present a short C code and a purely functional F# code of the solution to the well-known *n*-queens puzzle. Here is the C code first:

```
#include <stdio.h>
#include <stdlib.h>

typedef enum {FALSE, TRUE} BOOL;

#define N 8

int board[N + 1];

void print_array()
{
  int i;
  for (i = 1; i <= N; ++i)
    printf("%d ", board[i]);
  putchar('\n');
}

BOOL conflicting(int col, int row)
{
  int i;
  for (i = 1; i < col; ++i)
    if (board[i] == row || col - i == abs(row - board[i]))
```

```
      return TRUE;
  return FALSE;
}

void find_solutions(int col)
{
  static int num = 0;
  if (col > N)
  {
    printf("Solution #%02d: ", ++num);
    print_array();
  }
  else
  {
    int row;
    for (row = 1; row <= N; ++row)
      if (!conflicting(col, row))
      {
        board[col] = row;
        find_solutions(col + 1);
      }
  }
}

int main()
{
  find_solutions(1);
  return EXIT_SUCCESS;
}
```

And here is the F# code:

```
let N = 8

let conflicting col row (queen : int list) =
    let rec checkCol c =
        let r = queen.[c]
        c < col && (r = row || col - c = abs (row - r) || checkCol (c + 1))
    checkCol 0

let nextCol col newSolutions solution =
    seq {1 .. N}
        |> Seq.filter (fun row -> not (conflicting col row solution))
        |> Seq.fold (fun solutions row -> solutions @ [solution @ [row]]) newSolutions

let rec findSolutions col allSolutions =
    if col = N then
        allSolutions
    else
        findSolutions (col + 1) (allSolutions |> List.fold (nextCol col) [])

findSolutions 0 [[]]
    |> List.iteri (fun i solution -> printfn "Solution #%02d: %A" (i + 1) solution)
```

Both programs find the 92 possible solutions to the 8-queens problem, although they are not fully equivalent. The C code uses recursive backtracking, while the F# code is more like an optimized recursive breadth-first search, in which most of the work is done by built-in functions such as *Seq.fold*.

The C code works the following way: It takes a one-dimensional array of *N* elements (actually *N*+1 so we do not have to bother with the zero index), and calls the recursive function *find_solutions*, which tries to find an appropriate (non-conflicting) row for a queen in the next column of the table in a *for* loop. The index of the next column is stored in *col*, which is 1 at the beginning. If there is no such row, a backtracking is performed, i.e., *find_solutions* returns to its previous instance in the call stack (where the value of *col* was one less than its current value), and so it tries to find the next good row in the previous column inside the *for* loop. While we can find a good place for a queen in the current column, we continue calling *find_solutions* with an incre-

mented *col* value. When *col* reaches *N*+1, all *N* queens have been placed on the table, i.e., a solution is found. We print the solution, and continue with the search by backtracking (i.e., returning to the previous function in the call stack) until we return to the *main* function, which means that a backtracking was performed from the initial state.

The F# code uses a list of lists to store all the solutions. Each of the inner lists will finally contain *N* numbers with the row values for each column just like the array in the C version. At the beginning, we start with a one-element list of an empty list (`[[]]`), and then try to place a queen to all the possible rows in the column indicated by *col*. For example, when *col* is 0 and *allSolutions* is `[[]]`, the result of the expression `allSolutions |> List.fold (nextCol col) []` will be a list of 8 one-element list containing the numbers 1 to 8: `[[1], [2], [3], ...]`. After this, the *findSolutions* function is called recursively with an incremented *col* value, which results in a list containing only 2-element lists with all the possible layouts of two queens in two columns. This is repeated until *col* reaches *N*, when the resulting list will contain all possible solutions.

Of course, we could also have written here the recursive breadth-first search in C. The reason we chose backtracking instead is that it is much shorter because in backtracking, we only have to store the current path, and it is done for us by the call stack of the *find_solutions* function. In breadth-first search, however, we would have to keep track of the partial solutions (those in which one less columns are already filled than we are currently dealing with), which would require us to handle some kind of data structure (a linked list, for example). The F# version does this with the built-in *list* data type. Just for comparison, here is the C# implementation of the recursive breadth-first search, which resembles the most to the F# program (C# also has a built-in *List* data type):

```
using System;
using System.Text;
using System.Collections.Generic;

class Board
{
  public static int N = 8;

  private int[] board;
  private int col;

  public Board()
  {
    board = new int[N + 1];
    col = 1;
  }

  public Board(Board parent, int row)
  {
    board = (int[])parent.board.Clone();
    board[parent.col] = row;
    col = parent.col + 1;
  }

  public bool Conflicting(int row)
  {
    for (int i = 1; i < col; ++i)
      if (board[i] == row || col - i == Math.Abs(row - board[i]))
        return true;
    return false;
  }

  public override string ToString()
  {
    StringBuilder sb = new StringBuilder();
    for (int i = 1; i <= N; ++i)
      sb.Append(board[i] + " ");
```

```
    return sb.ToString();
  }
}

class Program
{
  static List<Board> findSolutions(int col, List<Board> allSolutions)
  {
    if (col > Board.N)
      return allSolutions;
    List<Board> newSolutions = new List<Board>();
    foreach (Board solution in allSolutions)
      for (int row = 1; row <= Board.N; ++row)
        if (!solution.Conflicting(row))
          newSolutions.Add(new Board(solution, row));
    return findSolutions(col + 1, newSolutions);
  }

  static void Main()
  {
    int num = 0;
    List<Board> initialList = new List<Board>();
    initialList.Add(new Board());
    foreach (Board solution in findSolutions(1, initialList))
      Console.WriteLine("Solution #{0:00}: {1}", ++num, solution);
  }
}
```

As you can see, the C# code is still much longer and, in my opinion, less expressive than the F# version of the very same algorithm.

Search Algorithms in Different Programming Languages

The first implementations of AI search algorithms were programmed using the first popular high-level imperative programming language, Fortran, and the first functional programming language, IPL. The General Problem Solver, created in 1959, was able to solve theoretically any formalized symbolic problems (Newell, Shaw, & Simon, 1959). Later, as newer and newer imperative programming languages (such as C or Pascal) dominated business computing, search algorithms were rewritten in a number of imperative languages. With the appearance of object-oriented paradigm, programmers had the possibility to easily create more abstract and general implementations of these algorithms.

Meanwhile, functional programming languages were undergoing vigorous development, too. Lisp, for example, was invented in 1958, and its variants (Common Lisp, Scheme, and Clojure among others) are still in use today. As an example, there is a Lisp implementation for solving the "farmer, wolf, goat, and cabbage problem" in Luger and Stubblefield (2009).

Prolog, designed specifically for logic programming in 1972, is a natural choice when it comes to programming AI search algorithms. If we would like to create the most concise implementation, we should use Prolog.

Why F#?

The problem with imperative languages lies in their verboseness. Even a very simple algorithm can take a lot of lines of code to implement. On the other hand, functional and logic programming languages require programmers to acquire a very special way of thinking about things, which may be appropriate for some sorts of real-world problems, but is unnatural for most problems. We think the solution is using multi-paradigm programming languages, like D, Python, C#, or F#. As we previously mentioned, modern imperative and object-oriented programming languages include some functional features. This way, developers may choose to do some kinds of computation functionally instead of the "traditional way." A good example to this is LINQ in C#.

F# is basically a functional language extended with imperative and object-oriented features for .NET interoperability. The programmer may write a program purely functionally, partly imperatively, using object-oriented tools, such as classes and objects, or by mixing any or all of these techniques (Petricek & Skeet, 2010; Syme, Granicz, & Cisternino, 2010). In F#, programmers may use object states, and this way we do not have to write mystic code, for example, for handling complex data structures. Another drawback of pure functional programming is the inefficiency of the executable code: copying data requires more memory and more runtime than just performing a small modification of existing data. Additionally, F# can help students realize that no single programming paradigm is best for everything. These are the reasons why F# seemed to be a good choice to implement the search algorithms, which we already had at our disposal in Java and C# (Kósa, 2009; Kósa & Pánovics, 2007).

A Class Hierarchy for Search Algorithms

Next, we present the class hierarchy that contains all the classes needed to implement various search algorithms for an arbitrary state-space represented problem. Figure 1 shows two abstract classes for the state-space representation itself (*State* and *Operator*) as well as a couple of *Node* classes, which represent the graph nodes used by the search algorithms.

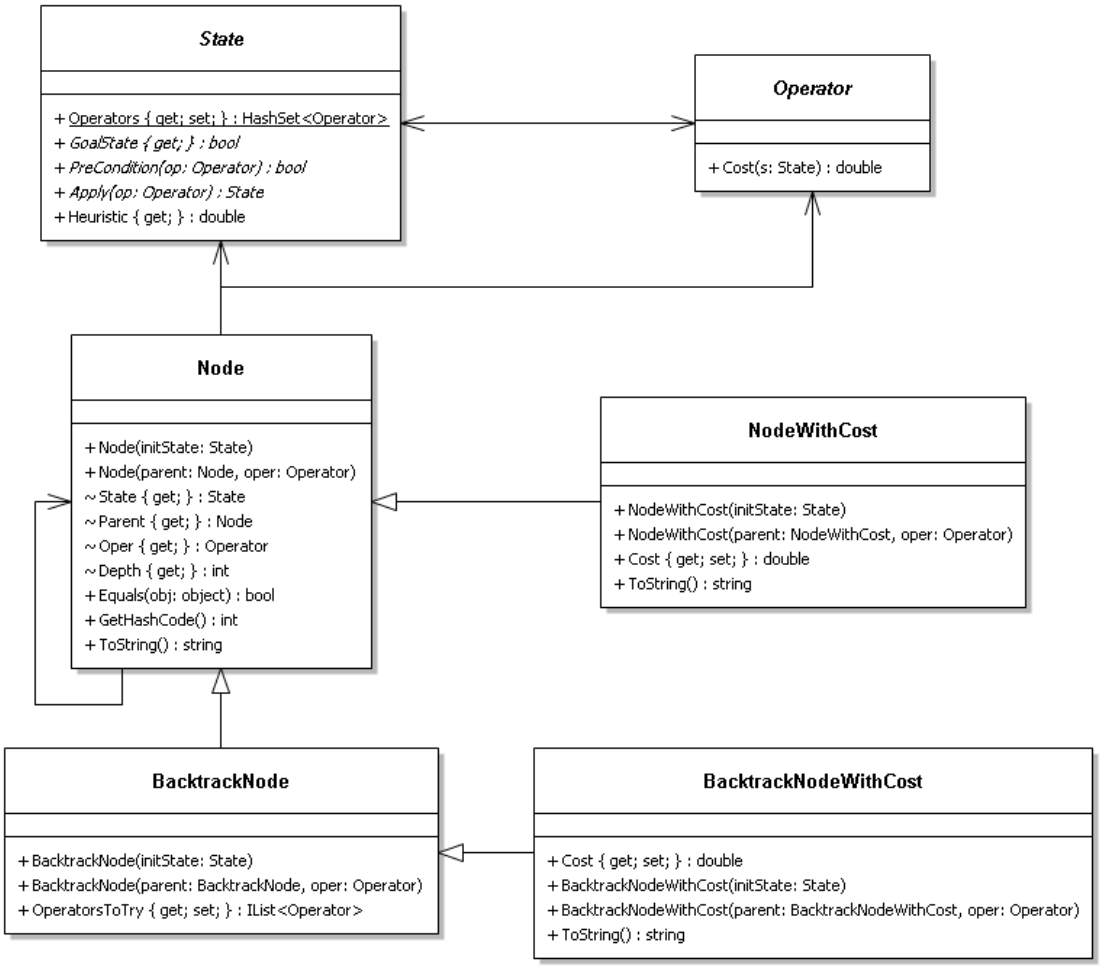

Figure 1: Classes representing the state-space and the graph nodes.

State is used as a base class for the classes representing the states of concrete problems, while the *Operator* class serves as a base class for the concrete operators, which transform our problem from one state to another. The members of these classes are the following:

- *Operators*: the set of all operators relevant to the problem.
- *GoalState*: true if the current state is a goal state.
- *PreCondition*: true if the argument operator is applicable to the current state.
- *Apply*: applies the argument operator to the current state and returns the resulting state.
- *Heuristic*: a heuristic value that is an estimation of the cost of reaching the nearest goal state from the current state.
- *Cost*: the cost of applying the current operator to the argument state.

Here is the F# code which defines these two classes:

```
type [<AbstractClass>] State() =
    static let operators = HashSet<Operator>()
    static member Operators = operators
    abstract GoalState : bool
    abstract PreCondition : Operator -> bool
    abstract Apply : Operator -> State
    abstract Heuristic : double
    default this.Heuristic = 0.0

and [<AbstractClass>] Operator() =
    abstract Cost : State -> double
    default this.Cost(_) = 1.0
```

As you can see, there is a default implementation of the *Heuristic* property so that we can use heuristic search algorithms (e.g., best-first search) even with states which do not override this property. Similarly, we gave a default implementation for the *Cost* method of the *Operator* class, this way ensuring that any operator may participate in a cost-based search (e.g., Dijkstra's algorithm; Dijkstra, 1959).

The four *Node* classes contain the following important members:

- *State*: the state represented by the node.
- *Parent*: the node to which an operator was applied to reach the current node.
- *Oper*: the operator that was applied to the parent node.
- *Depth*: the current node's depth in the spanning tree of the graph.
- *Cost*: the total cost of reaching the current node from the start node.
- *OperatorsToTry*: a list of operators applicable to the current node and not tried yet.

Node is used with non-cost-based graph search algorithms (e.g., breadth-first search), *NodeWithCost* is used with cost-based graph search algorithms (e.g., A algorithm), *BacktrackNode* is used with backtracking, and *BacktrackNodeWithCost* is used with the branch-and-bound algorithm. The latter two classes are actually used only in the C# version, which iteratively traverses the *OperatorsToTry* list. In contrast, the F# program uses recursion to go through the applicable operators (see the Appendix for details), so there is no need to explicitly store them in a list, and that is why there is no need for the *BacktrackNode* and *BacktrackNodeWithCost* classes, either.

Figure 2 shows another part of the UML class diagram which contains the classes representing some of the search algorithms and their relations to other classes.

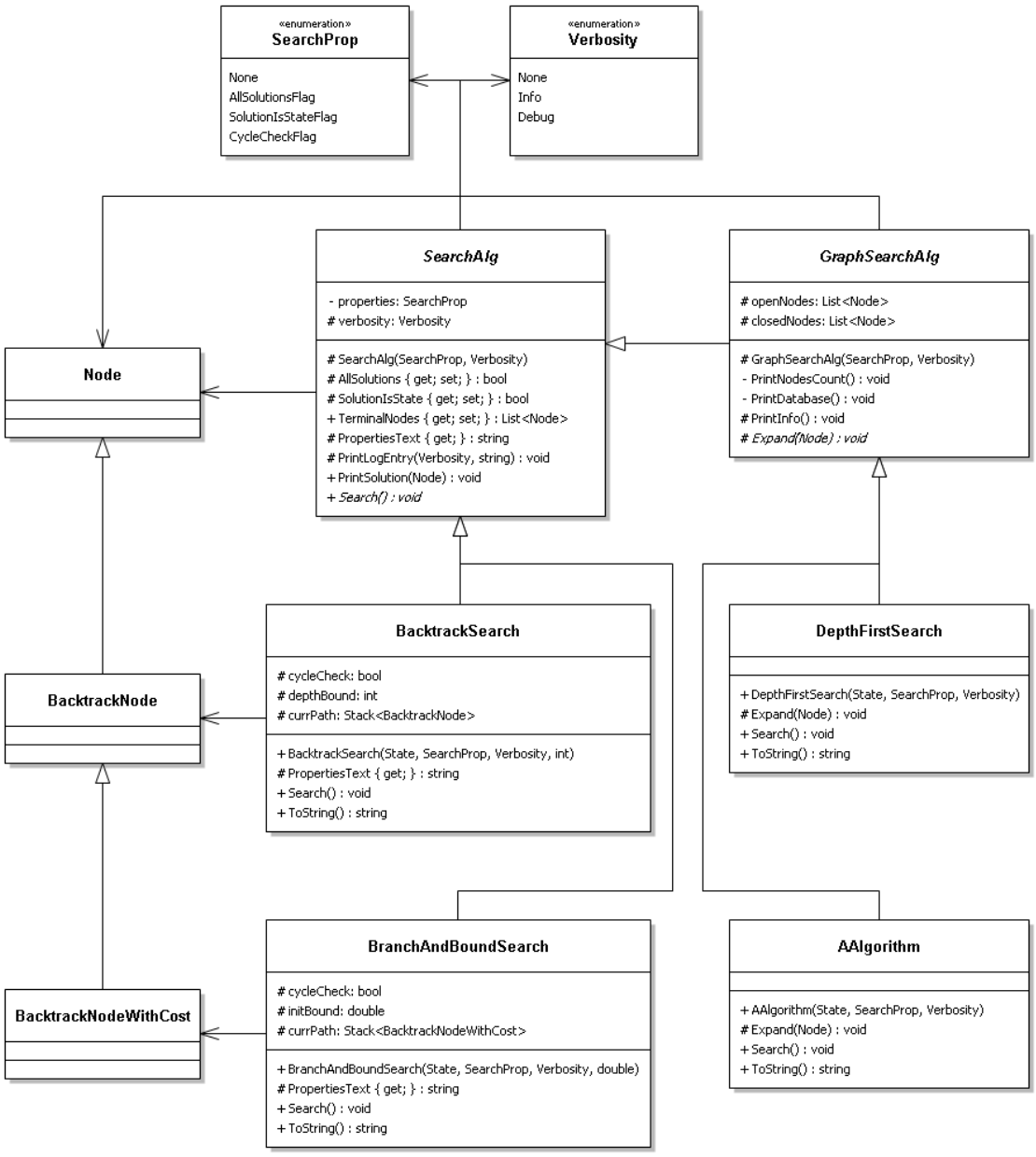

Figure 2: Classes representing some search algorithms.

Here you can see two enumeration types. *SearchProp* contains some flags which control the operation of the search algorithms. If *AllSolutionsFlag* is set, the algorithm will search for all solutions, otherwise it will stop when the first solution is found. If *SolutionIsStateFlag* is set, the algorithm considers the goal state as the solution, otherwise the solution is considered to be the operator sequence leading from the initial state to the goal state. *CycleCheckFlag* is used only with backtracking and branch-and-bound search. If it is set, the algorithm will check for cycles in the current path during the search, otherwise it may enter an infinite loop. *Verbosity* contains three verbosity levels which control the amount of information printed to the output during the search. The caller may pass as an argument any combination of the search property flags as well as one of the verbosity levels to the constructor of a particular search algorithm.

SearchAlg is an abstract class which is the base of all search algorithms and contains the following members:

- *AllSolutions* and *SolutionIsState* are two logical values which are relevant to all search algorithms. They provide easy access to two of the flags so that we do not have to mask the flags argument every time they are needed.
- *TerminalNodes*: a list of the terminal nodes found during the search.
- *PropertiesText*: a string representation of the search properties.
- *PrintLogEntry*: prints the given text to the output if the verbosity level of the search algorithm is greater than or equal to the given level.
- *PrintSolution*: prints the solution taken as an argument to the output.
- *Search*: an abstract method that does the actual work; it must be overridden by the concrete search algorithms.

In addition to these members, *BacktrackSearch* and *BranchAndBoundSearch* also store the current path as a stack of nodes, while *GraphSearchAlg* stores the open and closed nodes as lists of nodes. *GraphSearchAlg* also contains an abstract *Expand* method, which must be overridden by the concrete graph search algorithms.

As an example, here is the listing of the *DepthFirstSearch* class (you can find the full listing in the Appendix):

```
type DepthFirstSearch(initState, ?properties, ?verbosity) as this =
    inherit GraphSearchAlg(defaultArg properties SearchProp.None,
                           defaultArg verbosity Verbosity.Info)

    let verbosity = defaultArg verbosity Verbosity.Info

    do this.OpenNodes.Add(Node(initState))

    override this.Expand(node) =
        State.Operators
            |> Seq.filter (fun op -> node.State.PreCondition(op))
            |> Seq.iter (fun op ->
                let newNode = Node(node, op)
                if not (this.OpenNodes.Contains(newNode) ||
                        this.ClosedNodes.Contains(newNode)) then
                    this.OpenNodes.Insert(0, newNode))

    override this.Search() =
        this.PrintInfo()
        if this.OpenNodes.Count > 0 then
            let currNode = this.OpenNodes.[0]
            if currNode.State.GoalState then
                this.TerminalNodes.Add(currNode)
                if this.AllSolutions then
                    this.PrintLogEntry( Verbosity.Info, "Found a solution." )
                    this.OpenNodes.Remove(currNode) |> ignore
                    this.ClosedNodes.Add(currNode)
                    this.Search()
            else
                this.OpenNodes.Remove(currNode) |> ignore
                this.ClosedNodes.Add(currNode)
                this.Expand(currNode)
                this.Search()

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
            "Searching using depth-first search.\n"
                + this.PropertiesText
```

Let's examine this code in a little more detail. First, we declare the class itself and an implicit constructor with three parameters: the initial state and the optional search properties and verbosity level. In the next line, the base class is given with a constructor call, which initializes the open and closed nodes as empty lists. The only operation the constructor does is that it adds the start node (representing the initial state) to the (empty) list of open nodes.

After this, the abstract *Expand* method is overridden: it takes all the operators relevant to the problem, filters out those which are not applicable to the state represented by the node to be expanded, and applies all the remaining operators to this state by calling the constructor of *Node*. If the resulting node is not in the database, it is then added to the beginning of the list of open nodes.

The *Search* method is responsible for the control of the search algorithm. It is implemented here as a recursive function which terminates when there are no more open nodes left in the database. If there is at least one element in the list of open nodes, the first one is checked whether it is a goal state. If so, it is added to the list of terminal nodes. When we are searching for the first solution only, the algorithm terminates with just one terminal node, otherwise the current node is moved from the open nodes to the closed nodes, and the algorithm starts all over again. If the current node is not a terminal node, it is expanded after turning it into a closed node.

One difference between the C# and the F# versions of this code is in the *Search* method: the F# code uses tail recursion instead of a while loop. We can see a more conspicuous difference, however, in the *Expand* method: we get a more readable code using the forward pipe operator than using a conditional inside a *foreach* loop.

A Specific Problem

Of course, defining classes is not the area where we can get the most out of F#. We can gain much more if we implement a concrete problem with all the operator preconditions and applications. Let's consider another well-known puzzle as a simple example problem: the Towers of Hanoi. Figure 3 shows the two classes representing the states and operators of this particular problem.

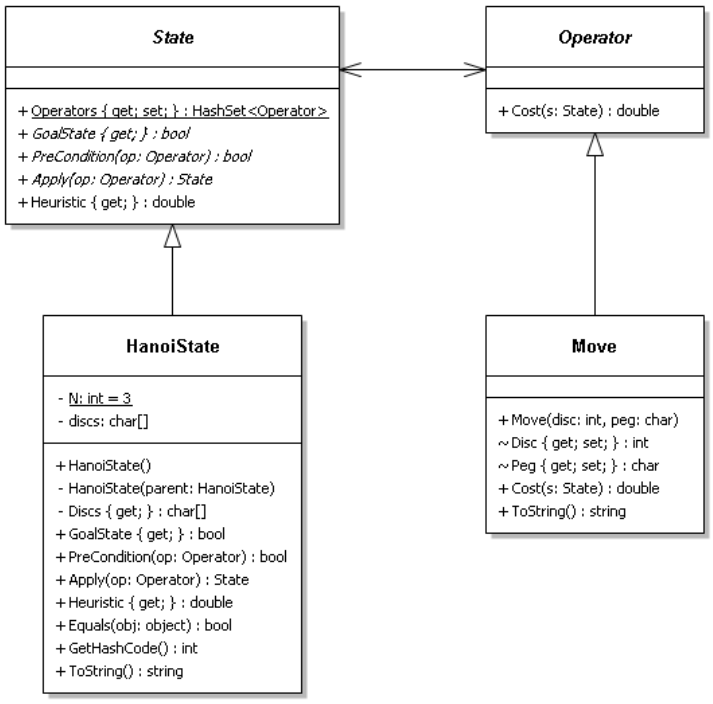

Figure 3: Classes representing a concrete problem.

The only extra member in *HanoiState* is *Discs*, which is an array of pegs where each disc can be found. *Move* comes with two members in addition to the derived ones: *Disc* tells us which disc to move, while *Peg* is the destination peg. Here is the full listing of these two classes in F#:

```
type Move(disc, peg) =
    inherit Operator()
    member this.Disc = disc
    member this.Peg = peg

    override this.ToString() =
        sprintf "HanoiMove[ disc=%d, peg='%c' ]" disc peg

    override this.Cost(_) = double disc

type HanoiState() =
    inherit State()

    static let N = 3
    let discs = Array.create N 'A'

    static do
        for disc in 1 .. N do
            for peg in 'A' .. 'C' do
                State.Operators.Add(Move(disc, peg)) |> ignore

    member private this.Discs = discs

    private new(parent : HanoiState) as this =
        HanoiState() then
            parent.Discs.CopyTo(this.Discs, 0)

    override this.GoalState =
        let rec allTheSame index =
            index >= discs.Length - 1
                || discs.[index] = discs.[index + 1]
                && allTheSame (index + 1)
        discs.[0] <> 'A' && allTheSame 0
```

```
override this.PreCondition(op) =
    match op with
        | :? Move as move ->
            let rec checkSmallerDiscs index =
                index >= move.Disc - 1
                    || discs.[index] <> discs.[move.Disc - 1]
                    && discs.[index] <> move.Peg
                    && checkSmallerDiscs (index + 1)
            checkSmallerDiscs 0 && discs.[move.Disc - 1] <> move.Peg
        | _ ->
            raise InvalidOperator

override this.Apply(op) =
    match op with
        | :? Move as move ->
            let newState = HanoiState(this)
            newState.Discs.[move.Disc - 1] <- move.Peg
            newState :> State
        | _ ->
            raise InvalidOperator

override this.Equals(other) =
    match other with
        | :? HanoiState as otherHanoiState ->
            this.Discs = otherHanoiState.Discs
        | _ ->
            false

override this.GetHashCode() =
    hash discs

override this.ToString() =
    let sb = System.Text.StringBuilder( "HanoiState[ discs=(" )
    for i in 0 .. N - 1 do
        if i > 0 then
            sb.Append(',') |> ignore
        sb.Append(discs.[i]) |> ignore
    sb.Append(") ]").ToString()

override this.Heuristic =
    let value1 = ref N
    let value2 = ref N
    for peg in discs do
        if peg = 'B' then
            decr value1
        elif peg = 'C' then
            decr value2
    double ( min !value1 !value2 )
```

The implementation of the *Move* operator is fairly straightforward. It has overridden the *Cost* method: the cost of moving a disc to another peg is proportional to its size, independently of the state to which the operator is applied.

The *HanoiState* class defines a one-dimensional array of characters for storing the pegs of each disc. The size of this array is *N* which stands for the number of discs in the problem. The smaller the index of an array element, the smaller the disc it represents. This seems to be a very efficient representation with very little memory required for storing a state.

The static constructor is responsible for creating all the possible operator instances and adding them to the static *Operators* property. The class has two constructors: the implicit constructor creates the initial state with each disc being on peg *A*, while the explicit constructor is actually a copy constructor, which creates a clone of the *HanoiState* object taken as a parameter.

The *GoalState* property first makes sure that the smallest disc is not on peg *A*, then checks whether all the discs are on the same peg (with the help of a recursive function). The skeleton of the *PreCondition* and *Apply* methods are the same: they both go through all the possible operator

types (currently there is only one: *Move*) and throw an exception if the argument is an operator of an unknown type. The *PreCondition* function has to make sure that none of the discs smaller than the one to be moved are on the source or the destination pegs and that the disc to be moved is not on the destination peg. The *Apply* method is very simple: it copies the current state and replaces the peg of the disc to be moved with the one determined by the operator. Finally, the *Heuristic* property determines the number of discs not being on peg *B* and the same for peg *C*, and returns the smaller of the two numbers because at least that many moves are required to reach a goal state.

There is not too much difference in code size between the C# and the F# versions of these two classes. The reason for this is the simplicity of the problem. The main difference is in the implementation of the *GoalState* property and the *PreCondition* method, both of which use recursion instead of loops in the F# version. With a more complex problem, we could see more improvement in the code of these two members because these are the two areas where logical formulae appear in the state-space representation, which (and especially existential and universal quantifications) can be much more conveniently implemented in F# than in C#.

Conclusion

As you may have noticed, the presented code is not purely functional. According to our experience, the code will not be shorter or more readable if we insist on writing purely functional code, i.e., without any side effects. However, the teachers of Artificial Intelligence seminars may decide whether they want to show a purely functional code to the students or implement the same algorithms using a little (or more) imperative code with just a little change in the existing code. That is why we think that F#, as a golden middle road between imperative and functional languages, seems to be a good alternative for presenting the AI search algorithms to the students.

The instructors of the Artificial Intelligence seminars at the University of Debrecen have been using our Java and C# implementation of the presented class hierarchy for more than five years now. We hope that in the future, they can make use of its F# version, too. It may be a good choice especially for students participating in a graduate program, who have previously learnt functional programming. Unfortunately, the core subjects of our undergraduate programs lack a thorough discussion of functional programming; it is only mentioned to students majoring Software Information Technology in the frame of the course *High Level Programming Languages 2* without any practice. Later, they can take an optional course titled *Programming Languages of Artificial Intelligence*, where they learn Clean, a purely functional language, but currently, there is no word about multi-paradigm languages. There is, however, a subject titled *New Programming Paradigms* in our Software Information Technology graduate program, and students learn F# in its laboratories. The first real feedback about the presence of F# in our education is that an agile graduate student chose this language to create an application for her master's thesis, which also includes some AI elements.

We plan to create a purely functional but reusable (not problem-specific) implementation of the most used search algorithms in F# and maybe other functional languages, too. First, we can get rid of the classes which are not used as a type and have only one instance, such as *DepthFirstSearch*. Next, the abstract *SearchAlg* and *GraphSearchAlg* classes would be replaced by simple functions calling other functions taken as parameter values. Finally, classes that act as data structures can be substituted with built-in F# types such as records, tuples, or lists as well as functions operating on them.

References

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik, 1*, 269-271.

King, D. J., & Launchbury, J. (1995). Structuring depth-first search algorithms in Haskell. *Proceedings of the 22nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 344-354.

Kósa, M. (2009). *Korszerű információtechnológiai módszerek bevezetése a mesterséges intelligencia oktatásába* [Introduction of modern information technology methods to the teaching of artificial intelligence]. Doctoral dissertation, University of Debrecen, Hungary. Retrieved July 19, 2012, from http://hdl.handle.net/2437/97347

Kósa, M., & Pánovics, J. (2007). Keresőalgoritmusok objektumorientált megközelítése a Mesterséges intelligencia tárgy bevezető kurzusán [Object-oriented approach of search algorithms at the introductory course of Artificial Intelligence]. *Proceedings of the 17th International Conference on Computers and Education*, 94-97.

Luger, G. F., & Stubblefield, W. A. (2009). *AI algorithms, data structures, and idioms in Prolog, Lisp, and Java*. Boston: Pearson Education.

Newell, A., Shaw, J. C., & Simon, H. A. (1959). Report on a general problem-solving program. *Proceedings of the International Conference on Information Processing*, 256-264.

Petricek, T., & Skeet, J. (2010). *Real-world functional programming*. Greenwich: Manning Publications.

Syme, D., Granicz, A., & Cisternino, A. (2010). *Expert F# 2.0*. New York: Apress.

Appendix. The F# Listing of the Full Class Hierarchy

```
//StateSpace.fs
namespace StateSpace

open System.Collections.Generic

type [<AbstractClass>] State() =
    static let operators = HashSet< Operator >()
    static member Operators = operators
    abstract GoalState : bool
    abstract PreCondition : Operator -> bool
    abstract Apply : Operator -> State
    abstract Heuristic : double
    default this.Heuristic = 0.0

and [<AbstractClass>] Operator() =
    abstract Cost : State -> double
    default this.Cost( _ ) = 1.0

exception InvalidOperator


//Node.fs
namespace SearchAlg

open System
open System.Collections.Generic
open StateSpace

type Node =
    val private state  : State
    val private parent : Node option
    val private oper   : Operator option
    val private depth  : int

    member this.State  = this.state
    member this.Parent = this.parent
    member this.Oper   = this.oper
    member this.Depth  = this.depth
```

```
    new( initState : State ) =
        { state = initState; parent = None; oper = None; depth = 0 }

    new( parent : Node, oper : Operator ) =
        { state  = parent.state.Apply( oper )
          parent = Some parent
          oper   = Some oper
          depth  = parent.depth + 1 }

    override this.Equals( other ) =
        match other with
            | :? Node as otherNode ->
                this.state.Equals( otherNode.state )
            | _ ->
                false

    override this.GetHashCode() =
        hash this.state

    override this.ToString() =
        let s =
            sprintf "%s%O (depth=%d"
                ( if this.oper = None then "" else this.oper.Value.ToString() + " => " )
                this.state this.depth
        let heurProp = this.state.GetType().GetProperty( "Heuristic" )
        if heurProp.DeclaringType = heurProp.ReflectedType then
            s + sprintf ", heuristic=%g)" this.state.Heuristic
        else
            s + ")"

type NodeWithCost =
    inherit Node

    val private cost : double
    member this.Cost = this.cost

    new( initState : State ) =
        { inherit Node( initState ); cost = 0.0 }

    new( parent : NodeWithCost, oper : Operator ) =
        { inherit Node( parent, oper );
          cost = parent.cost + oper.Cost( parent.State ) }

    override this.ToString() =
        base.ToString() + ", cost=" + this.cost.ToString()


//SearchAlg.fs
namespace SearchAlg

open System
open System.Collections.Generic

[<Flags>]
type SearchProp =
    | None               = 0b00000000
    | AllSolutionsFlag   = 0b00000001
    | SolutionIsStateFlag = 0b00000010
    | CycleCheckFlag     = 0b00000100

type Verbosity =
    | None  = 0
    | Info  = 1
    | Debug = 2

[<AbstractClass>]
type SearchAlg( ?properties, ?verbosity ) =
    let properties = defaultArg properties SearchProp.None
    let verbosity = defaultArg verbosity Verbosity.Info
    let allSolutions = properties &&& SearchProp.AllSolutionsFlag <> SearchProp.None
```

```
    let solutionIsState = properties &&& SearchProp.SolutionIsStateFlag <>
                          SearchProp.None
    let terminalNodes = List< Node >()

    member this.AllSolutions = allSolutions
    member this.SolutionIsState = solutionIsState
    member this.TerminalNodes = terminalNodes

    abstract PropertiesText : string
    default this.PropertiesText =
        ( if allSolutions then
              "Searching for all solutions.\n"
          else
              "Searching for the first solution.\n" ) +
        ( if solutionIsState then
              "The goal state is considered to be the solution.\n"
          else
              "The operator sequence leading to the goal state is the solution.\n" ) +
        "Verbosity level: " + verbosity.ToString() + "\n"

    member this.PrintLogEntry( minLevel, entry ) =
        if verbosity >= minLevel then
            printfn "%s" entry

    member this.PrintSolution( terminal : Node option ) =
        if solutionIsState then
            try
                printfn "%O" terminal.Value.State
            with
                :? NullReferenceException -> printfn "Null as a solution???"
        else
            if terminal.IsSome then
                this.PrintSolution( terminal.Value.Parent )
                printfn "%O" terminal.Value

    abstract Search : unit -> unit


//Backtrack.fs
namespace SearchAlg

open System.Collections.Generic
open StateSpace

exception InvalidBound

type BacktrackSearch( initState, ?properties, ?verbosity, ?depthBound ) =
    inherit SearchAlg( defaultArg properties SearchProp.None,
                       defaultArg verbosity Verbosity.Info )

    let properties = defaultArg properties SearchProp.None
    let verbosity = defaultArg verbosity Verbosity.Info
    let depthBound = defaultArg depthBound 0
    let cycleCheck = properties &&& SearchProp.CycleCheckFlag <> SearchProp.None
    let currPath = Stack< Node >()

    do
        if depthBound < 0 then
            raise InvalidBound
        currPath.Push( Node( initState ) )

    override this.PropertiesText =
        base.PropertiesText +
            ( if cycleCheck then
                  "Cycle check is on.\n"
              else
                  "Cycle check is off.\n" ) +
            ( if depthBound > 0 then
                  "Depth bound: " + depthBound.ToString() + "\n"
              else
                  "Depth bound check is off.\n" )
```

```
    override this.Search() =
        let currNode = currPath.Peek()
        let depthText =
            if depthBound > 0 then
                sprintf " (depth=%d)" currNode.Depth
            else
                ""
        if currNode.State.GoalState then
            this.PrintLogEntry( Verbosity.Debug,
                sprintf "Current state: %O%s" currNode.State depthText )
            if not ( this.SolutionIsState &&
                this.TerminalNodes.Contains( currNode ) ) then
                this.TerminalNodes.Add( currNode )
            if this.AllSolutions then
                this.PrintLogEntry( Verbosity.Info,
                    "Found a solution, backtracking." )
                currPath.Pop() |> ignore
        elif depthBound > 0 && currNode.Depth = depthBound then
            this.PrintLogEntry( Verbosity.Debug,
                sprintf "Current state: %O%s" currNode.State depthText )
            this.PrintLogEntry( Verbosity.Info,
                "Reached depth bound, backtracking." )
            currPath.Pop() |> ignore
        else
            State.Operators
                |> Seq.filter ( fun op -> currNode.State.PreCondition( op ) )
                |> Seq.takeWhile ( fun _ ->
                    this.AllSolutions || this.TerminalNodes.Count = 0 )
                |> Seq.iter ( fun op ->
                    this.PrintLogEntry( Verbosity.Debug,
                        sprintf "Current state: %O%s" currNode.State depthText )
                    this.PrintLogEntry( Verbosity.Debug,
                        sprintf "Applying operator: %O" op )
                    let newNode = Node( currNode, op )
                    this.PrintLogEntry( Verbosity.Debug,
                        sprintf "New state: %O" newNode.State )
                    if cycleCheck && currPath.Contains( newNode ) then
                        this.PrintLogEntry( Verbosity.Info, "Found a cycle." )
                    else
                        currPath.Push( newNode )
                        this.Search() )
            if this.AllSolutions || this.TerminalNodes.Count = 0 then
                this.PrintLogEntry( Verbosity.Debug,
                    sprintf "Current state: %O%s" currNode.State depthText )
                this.PrintLogEntry( Verbosity.Info,
                    "No more applicable operators, backtracking." )
                currPath.Pop() |> ignore

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
            "Searching using backtracking.\n" + this.PropertiesText

type BranchAndBoundSearch( initState, ?properties, ?verbosity, ?initBound ) =
    inherit SearchAlg( defaultArg properties SearchProp.None,
                       defaultArg verbosity Verbosity.Info )

    let properties = defaultArg properties SearchProp.None
    let verbosity = defaultArg verbosity Verbosity.Info
    let initBound = defaultArg initBound 0.0
    let cycleCheck = properties &&& SearchProp.CycleCheckFlag <> SearchProp.None
    let currPath = Stack< NodeWithCost >()

    do currPath.Push( NodeWithCost( initState ) )

    override this.PropertiesText =
        base.PropertiesText +
            ( if cycleCheck then
                  "Cycle check is on.\n"
```

```
          else
              "Cycle check is off.\n" ) +
        ( if initBound > 0.0 then
              "Initial cost bound: " + initBound.ToString() + "\n"
          else
              "No initial cost bound.\n" )

override this.Search() =
    let rec search currBound =
        let currNode = currPath.Peek()
        if currNode.State.GoalState
            && ( currBound <= 0.0 || currNode.Cost <= currBound ) then
            this.PrintLogEntry( Verbosity.Debug,
                sprintf "Current state: %O, cost: %g" currNode.State currNode.Cost )
            this.PrintLogEntry( Verbosity.Info,
                sprintf "Found a solution with cost %g, backtracking."
                currNode.Cost )
            if currNode.Cost < currBound then
                this.PrintLogEntry( Verbosity.Info,
                    sprintf "New cost bound: %g" currNode.Cost )
                this.TerminalNodes.Clear()
            if this.TerminalNodes.Count = 0 || this.AllSolutions &&
                not ( this.SolutionIsState &&
                this.TerminalNodes.Contains( currNode ) ) then
                this.TerminalNodes.Add( currNode )
            currPath.Pop() |> ignore
            currNode.Cost
        elif currBound > 0.0 && currNode.Cost >= currBound then
            this.PrintLogEntry( Verbosity.Debug,
                sprintf "Current state: %O, cost: %g" currNode.State currNode.Cost )
            this.PrintLogEntry( Verbosity.Info,
                "Reached cost bound, backtracking." )
            currPath.Pop() |> ignore
            currBound
        else
            let newBound =
                State.Operators
                    |> Seq.filter ( fun op -> currNode.State.PreCondition( op ) )
                    |> Seq.fold ( fun bound op ->
                        this.PrintLogEntry( Verbosity.Debug,
                            sprintf "Current state: %O, cost: %g"
                                currNode.State currNode.Cost )
                        this.PrintLogEntry( Verbosity.Debug,
                            sprintf "Applying operator: %O" op )
                        let newNode = NodeWithCost( currNode, op )
                        this.PrintLogEntry( Verbosity.Debug,
                            sprintf "New state: %O" newNode.State )
                        if cycleCheck && currPath.Contains( newNode ) then
                            this.PrintLogEntry( Verbosity.Info, "Found a cycle." )
                            bound
                        else
                            currPath.Push( newNode )
                            search bound ) currBound
            this.PrintLogEntry( Verbosity.Debug,
                sprintf "Current state: %O, cost: %g" currNode.State currNode.Cost )
            this.PrintLogEntry( Verbosity.Info,
                "No more applicable operators, backtracking." )
            currPath.Pop() |> ignore
            newBound

    this.PrintLogEntry( Verbosity.Debug, sprintf "Initial cost bound: %g" initBound )
    search initBound |> ignore

override this.ToString() =
    if verbosity = Verbosity.None then
        ""
    else
        "Searching using branch and bound algorithm.\n" + this.PropertiesText
```

```
//GraphSearchAlg.fs
namespace SearchAlg

open System.Collections.Generic
open StateSpace

[<AbstractClass>]
type GraphSearchAlg( ?properties, ?verbosity ) =
    inherit SearchAlg( defaultArg properties SearchProp.None,
                       defaultArg verbosity Verbosity.Info )

    let verbosity = defaultArg verbosity Verbosity.Info
    let openNodes = List< Node >()
    let closedNodes = List< Node >()

    member this.OpenNodes = openNodes
    member this.ClosedNodes = closedNodes

    member this.PrintInfo() =
        let printNodesCount () =
            printfn "Open nodes: %d, closed nodes: %d." this.OpenNodes.Count
                this.ClosedNodes.Count

        let printDatabase () =
            printfn "Open nodes:"
            for node in openNodes do
                printfn "%O" node
            printfn "Closed nodes:"
            for node in closedNodes do
                printfn "%O" node
            printfn ""

        if verbosity = Verbosity.Info then
            printNodesCount ()
        elif verbosity = Verbosity.Debug then
            printDatabase ()

    abstract Expand : Node -> unit

type BreadthFirstSearch( initState, ?properties, ?verbosity ) as this =
    inherit GraphSearchAlg( defaultArg properties SearchProp.None,
                            defaultArg verbosity Verbosity.Info )

    let verbosity = defaultArg verbosity Verbosity.Info

    do this.OpenNodes.Add( Node( initState ) )

    override this.Expand( node ) =
        State.Operators
            |> Seq.filter ( fun op -> node.State.PreCondition( op ) )
            |> Seq.iter ( fun op ->
                let newNode = Node( node, op )
                if not ( this.OpenNodes.Contains( newNode ) ||
                         this.ClosedNodes.Contains( newNode ) ) then
                    this.OpenNodes.Add( newNode ) )

    override this.Search() =
        this.PrintInfo()
        if this.OpenNodes.Count > 0 then
            let currNode = this.OpenNodes.[ 0 ]
            if currNode.State.GoalState then
                this.TerminalNodes.Add( currNode )
                if this.AllSolutions then
                    this.PrintLogEntry( Verbosity.Info, "Found a solution." )
                    this.OpenNodes.Remove( currNode ) |> ignore
                    this.ClosedNodes.Add( currNode )
                    this.Search()
            else
                this.OpenNodes.Remove( currNode ) |> ignore
                this.ClosedNodes.Add( currNode )
                this.Expand( currNode )
```

```
                this.Search()

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
            "Searching using breadth-first search.\n" + this.PropertiesText

type DepthFirstSearch( initState, ?properties, ?verbosity ) as this =
    inherit GraphSearchAlg( defaultArg properties SearchProp.None,
                            defaultArg verbosity Verbosity.Info )

    let verbosity = defaultArg verbosity Verbosity.Info

    do this.OpenNodes.Add( Node( initState ) )

    override this.Expand( node ) =
        State.Operators
            |> Seq.filter ( fun op -> node.State.PreCondition( op ) )
            |> Seq.iter ( fun op ->
                let newNode = Node( node, op )
                if not ( this.OpenNodes.Contains( newNode ) ||
                         this.ClosedNodes.Contains( newNode ) ) then
                    this.OpenNodes.Insert( 0, newNode ) )

    override this.Search() =
        this.PrintInfo()
        if this.OpenNodes.Count > 0 then
            let currNode = this.OpenNodes.[ 0 ]
            if currNode.State.GoalState then
                this.TerminalNodes.Add( currNode )
                if this.AllSolutions then
                    this.PrintLogEntry( Verbosity.Info, "Found a solution." )
                    this.OpenNodes.Remove( currNode ) |> ignore
                    this.ClosedNodes.Add( currNode )
                    this.Search()
            else
                this.OpenNodes.Remove( currNode ) |> ignore
                this.ClosedNodes.Add( currNode )
                this.Expand( currNode )
                this.Search()

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
            "Searching using depth-first search.\n" + this.PropertiesText

type DijkstraSearch( initState, ?properties, ?verbosity ) as this =
    inherit GraphSearchAlg( defaultArg properties SearchProp.None,
                            defaultArg verbosity Verbosity.Info )

    let verbosity = defaultArg verbosity Verbosity.Info

    do this.OpenNodes.Add( NodeWithCost( initState ) )

    override this.Expand( node ) =
        State.Operators
            |> Seq.filter ( fun op -> node.State.PreCondition( op ) )
            |> Seq.iter ( fun op ->
                let newNode = NodeWithCost( node :?> NodeWithCost, op )
                let index = this.OpenNodes.IndexOf( newNode )
                if index <> -1 then
                    let oldNode = this.OpenNodes.[ index ] :?> NodeWithCost
                    if newNode.Cost < oldNode.Cost then
                        this.OpenNodes.Remove( oldNode ) |> ignore
                        this.OpenNodes.Add( newNode )
                elif not ( this.ClosedNodes.Contains( newNode ) ) then
                    this.OpenNodes.Add( newNode ) )

    override this.Search() =
```

```
        this.PrintInfo()
        if this.OpenNodes.Count > 0 then
            let currNode = this.OpenNodes.[ 0 ] :?> NodeWithCost
            if not ( this.TerminalNodes.Count > 0 &&
                currNode.Cost > ( this.TerminalNodes.[ 0 ] :?> NodeWithCost ).Cost ) then
                if currNode.State.GoalState then
                    this.TerminalNodes.Add( currNode )
                    if this.AllSolutions then
                        this.PrintLogEntry( Verbosity.Info, "Found a solution." )
                        this.OpenNodes.Remove( currNode ) |> ignore
                        this.ClosedNodes.Add( currNode )
                        this.Search()
                else
                    this.OpenNodes.Remove( currNode ) |> ignore
                    this.ClosedNodes.Add( currNode )
                    this.Expand( currNode )
                    this.OpenNodes.Sort(
                        { new IComparer< Node > with
                            member this.Compare( n1, n2 ) =
                                ( n1 :?> NodeWithCost ).Cost.CompareTo(
                                    ( n2 :?> NodeWithCost ).Cost ) } )
                    this.Search()

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
            "Searching using Dijkstra's algorithm.\n" + this.PropertiesText

type BestFirstSearch( initState, ?properties, ?verbosity ) as this =
    inherit GraphSearchAlg( defaultArg properties SearchProp.None,
                            defaultArg verbosity Verbosity.Info )

    let verbosity = defaultArg verbosity Verbosity.Info

    do this.OpenNodes.Add( Node( initState ) )

    override this.Expand( node ) =
        State.Operators
            |> Seq.filter ( fun op -> node.State.PreCondition( op ) )
            |> Seq.iter ( fun op ->
                let newNode = Node( node, op )
                if not ( this.OpenNodes.Contains( newNode ) ||
                         this.ClosedNodes.Contains( newNode ) ) then
                    this.OpenNodes.Add( newNode ) )

    override this.Search() =
        this.PrintInfo()
        if this.OpenNodes.Count > 0 then
            let currNode = this.OpenNodes.[ 0 ]
            if currNode.State.GoalState then
                this.TerminalNodes.Add( currNode )
                if this.AllSolutions then
                    this.PrintLogEntry( Verbosity.Info, "Found a solution." )
                    this.OpenNodes.Remove( currNode ) |> ignore
                    this.ClosedNodes.Add( currNode )
                    this.Search()
            else
                this.OpenNodes.Remove( currNode ) |> ignore
                this.ClosedNodes.Add( currNode )
                this.Expand( currNode )
                this.OpenNodes.Sort(
                    { new IComparer< Node > with
                        member this.Compare( n1, n2 ) =
                            n1.State.Heuristic.CompareTo( n2.State.Heuristic ) } )
                this.Search()

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
```

```
            "Searching using best-first search.\n" + this.PropertiesText

type AAlgorithm( initState, ?properties, ?verbosity ) as this =
    inherit GraphSearchAlg( defaultArg properties SearchProp.None,
                            defaultArg verbosity Verbosity.Info )

    let verbosity = defaultArg verbosity Verbosity.Info

    do this.OpenNodes.Add( NodeWithCost( initState ) )

    override this.Expand( node ) =
        State.Operators
            |> Seq.filter ( fun op -> node.State.PreCondition( op ) )
            |> Seq.iter ( fun op ->
                let newNode = NodeWithCost( node :?> NodeWithCost, op )
                let index = this.OpenNodes.IndexOf( newNode )
                if index <> -1 then
                    let oldNode = this.OpenNodes.[ index ] :?> NodeWithCost
                    if newNode.Cost < oldNode.Cost then
                        this.OpenNodes.Remove( oldNode ) |> ignore
                        this.OpenNodes.Add( newNode )
                else
                    let index = this.ClosedNodes.IndexOf( newNode )
                    if index <> -1 then
                        let oldNode = this.ClosedNodes.[ index ] :?> NodeWithCost
                        if newNode.Cost < oldNode.Cost then
                            this.ClosedNodes.Remove( oldNode ) |> ignore
                            this.OpenNodes.Add( newNode )
                    else
                        this.OpenNodes.Add( newNode ) )

    override this.Search() =
        this.PrintInfo()
        if this.OpenNodes.Count > 0 then
            let currNode = this.OpenNodes.[ 0 ] :?> NodeWithCost
            if currNode.State.GoalState then
                this.TerminalNodes.Add( currNode )
                if this.AllSolutions then
                    this.PrintLogEntry( Verbosity.Info, "Found a solution." )
                    this.OpenNodes.Remove( currNode ) |> ignore
                    this.ClosedNodes.Add( currNode )
                    this.Search()
            else
                this.OpenNodes.Remove( currNode ) |> ignore
                this.ClosedNodes.Add( currNode )
                this.Expand( currNode )
                this.OpenNodes.Sort(
                    { new IComparer< Node > with
                        member this.Compare( n1, n2 ) =
                            let f1 = ( n1 :?> NodeWithCost ).Cost + n1.State.Heuristic
                            let f2 = ( n2 :?> NodeWithCost ).Cost + n2.State.Heuristic
                            f1.CompareTo( f2 ) } )
                this.Search()

    override this.ToString() =
        if verbosity = Verbosity.None then
            ""
        else
            "Searching using the A algorithm.\n" + this.PropertiesText
```

Biography

János Pánovics is an assistant lecturer at the University of Debrecen, Hungary, where he received his master's degree in Computer Science (IT).

His general research interests include programming languages (both low-level and high-level), programming paradigms (imperative, object-oriented, functional, and logic), artificial intelligence, database technologies, and IT education. He has had teaching experience in various fields of IT, including subjects like Assembly Languages, Computer Architectures, High-Level Programming Languages, Data Structures and Algorithms, Database Systems, and Artificial Intelligence.