2020 INFORINO
Moscow, Russia
14-17 April, 2020

Software Environment for Studying Intelligent Anytime Heuristic Search Methods

Speacker:
Ilia
Poliushkin
NRU "MPEI"

Authors:

Alexander P. Eremeev

Ilia A. Poliushkin

Nikolai A. Paniavin

Department of Applied Mathematics and Artificial Intelligence

National Research University "MPEI"

IEEE

NATIONAL RESEARCH UNIVERSITY

CROC

Abstract

Anytime algorithms are widely used in real-time intelligent systems because they offer a tradeoff between solution quality and solving time. This property of them is valuable when the first solution can be obtained much faster than obtaining an optimal solution with the classical algorithm. In this paper, we describe several frameworks for making anytime-algorithms from classical heuristic searches and introducing a C ++ implementation of these algorithms useful for demonstrating to the students the concept of an anytime heuristic search, its use, properties and tuning. These algorithms are studied by students who are studying in the master's program in the field of “Applied mathematics and Informatics” when studying methods and systems of artificial intelligence (discipline "Methods and software tools for decision support").

Introduction

Anytime algorithms are algorithms, which result’s quality monotonously improves with increasing algorithm’s runtime. Such algorithms acquire some solution in a short period and then improve it over time. You can stop them at any time and get the best solution found at that moment.

Using algorithms of this kind allows us to find tradeoff between the quality of the solution found and the time taken to find it. This can be useful in real-time intelligent systems. The amount time to find a solution in this system is limited. The solution with worse quality found on time is much more valuable than the optimal solution found by the time it is already useless. For example, because the system has already changed its state and a completely different solution is required. Alternatively, the original goal has become unreachable.

Obviously, if there is a procedure for evaluating the quality of a solution (or, at least, comparing two solutions), an algorithm that non-monotonously improves its solution can easily be made monotonic.

This fact allows us to consider such widely used algorithms as iterative numerical methods as an example of anytime algorithms. For example, a bisection method or a simple iteration method.

Introduction

The present report reviews anytime algorithms for state space search with the intention of mastering them by students and then using them in the design of prototypes of intelligent decision support systems in the study of the discipline "Methods and Decision Support Systems", as well as in preparing the master's thesis.

It is said that the ***problem of searching in the state space*** is stated when the following five are given:

- Set of states;
- Initial state — the state of the system at the initial moment;
- Move function — a description of possible moves from one state to another;
- Target check — some algorithm that allows us to determine whether a given state is a target state;
- Path cost function — a function that assigns a certain cost to each sequence of moves between states.

A ***solution for the search problem in a state space*** is a sequence of transitions leading from an initial state to a target state. The path cost is a sum of costs of all moves. A cost of each transition is a non-negative value.

A quality of a solution can be defined as a ratio of a cost of an optimal solution to a cost of a current solution.

Conversion Methods

To solve a search problem in the state space, there are search algorithms based on a heuristic function as A*, IDA*, RBFS and others. None of these algorithms is anytime. However, there are number of methods that allow us to convert these algorithms into anytime searches.

All these methods require a pair of heuristic functions: admissible and non-admissible. Let us remind that ***a heuristic function is called admissible*** if it does not overestimate the cost of the path from any vertex to the target. A heuristic function that does not have this property is called ***non-admissible***. The chosen non-admissible heuristic function should provide a solution in a shorter period than the admissible one. We can use a weighted admissible heuristic function, for example.

The first of these methods is called ***restarting***. It consists of repeating the search after an algorithm finds the first solution using a non-admissible heuristic function. However, algorithm discards vertices, an admissible estimate of the cost of the solution passing through which is no lower than the cost of the previously found solution. This allows us to find a solution with a cost lower than the cost of previously found solution. This procedure is repeated while another solution can be found.

The second method, called ***continuing***, is similar to restarting. The difference is that after finding the next solution, the lists of open and closed vertices are not cleared. That is, the search does not restart, but continues from the same place.

Conversion Methods

The third method, called ***repairing***, is similar to the continuing method, but has two significant differences.

The first difference is that when the algorithm finds a shorter path to the vertex that is already in the open or closed lists, it does not immediately reopen this vertex, but put it into a special INCONS list. The vertices from this list are moved to the open list after finding the next solution. This addition allows the algorithm to find the next solution quickly, since it does not take time to open the same vertices for many times at one search iteration. The solution obtained at a certain step will have lesser quality, but the full potential of reopening these vertices will be used in the next iteration of the search. It should be noted that if several new paths were found to the vertex, only one, namely the best one, would be used. This allows us to speed up the search significantly in some types of graphs, but it can also significantly slow it down in other cases.

The second difference is that the parameters of the algorithm change as each new solution is found. For example, the weight of the heuristic function may decrease, tending to one, with an improvement of the solution.

These two modifications can be applied to the restarting method, either together or separately.

Anytime A* Algorithm

A* algorithm uses the following formula to estimate the cost of a path passing through vertex *n*:

$f(n)=g(n)+h(n)$,

where $g(n)$ is the cost of the path traveled from the initial vertex, and $h(n)$ is an admissible estimate of the cost of the path remaining to the goal.

As an inadmissible estimate, we will use the formula:

$f(n)=g(n)+w*h(n)$,

where $w > 1$ is the weight of the heuristic function.

All three methods can be applied to the A* algorithm with this pair of heuristic functions.

It is proved that if the heuristic function h(n) is admissible, then the solution found using the weighted heuristic function has a cost no more than w times the optimal solution cost. Thus, it is possible to assess the quality of the first solution found by restarting and continuing A* algorithms.

It should be noted that this estimate will not be valid for the repairing A* algorithm, since it also changes the process of finding the first solution.

Implementation of Algorithms and Applications

Algorithms are implemented in C++ using an object-oriented approach.

The class interfaces are declared in such a way that the application of search algorithms to various problems does not require changing the source code of the algorithm classes.

All classes of search algorithms are inherited from the base *SearchBase* class, which makes it possible to unify access to various algorithms.

The *SearchBase* class is parametric and takes two parameters. The first parameter *G* is the representation class for the graph of the problem's state space. The second parameter *S* is the representation state class of the specific state.

Class *G* must have the following member functions:

double hFunction (S & s) is an admissible heuristic function for estimating the cost of a path passing through the state of *s* ;

double cFunction (S & s1, S & s1) returns transition cost between two states of *s1* and *s2* ;

bool isGoal (S & s) checks if the state *s* is in the set of target states ;

std::list<S> adjacent(S & s) returns a list of states that are directly accessible from the *s* state.

Implementation of Algorithms and Applications

Class *S* must have a comparison operator “less”.

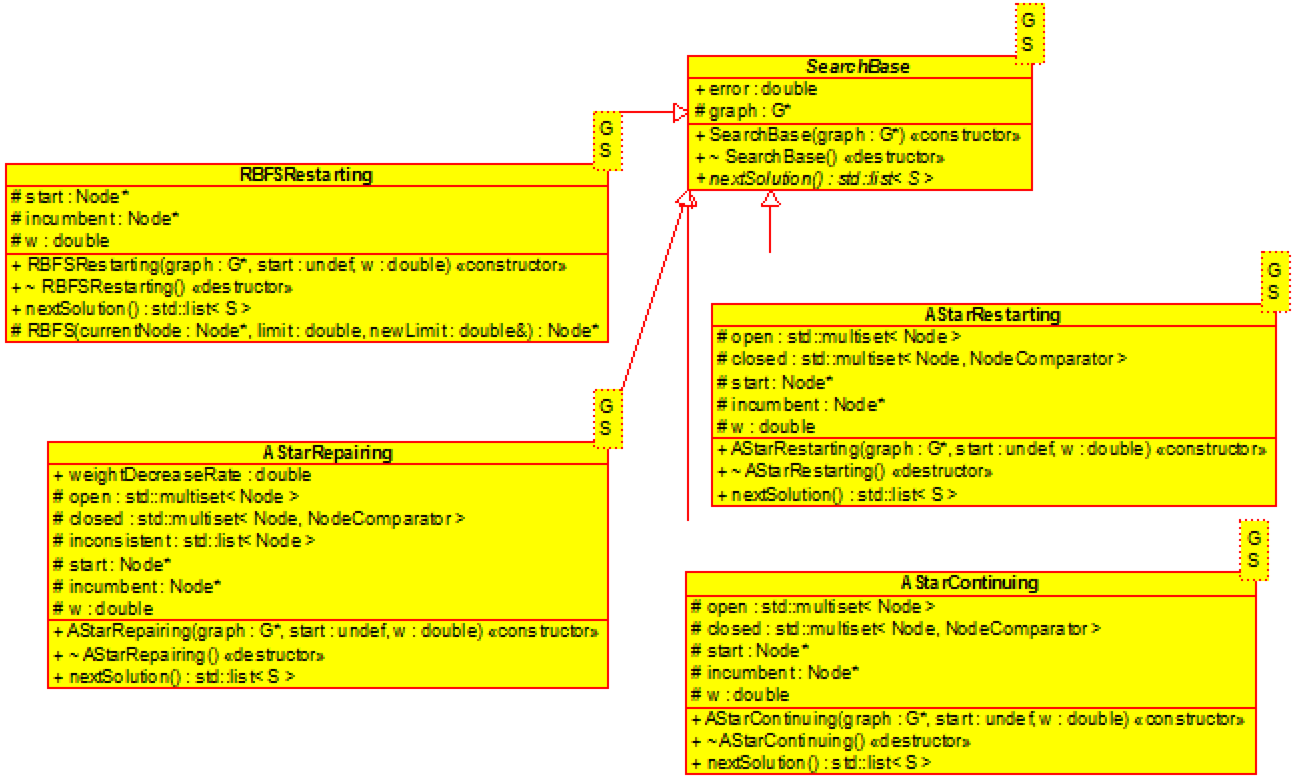
Solutions are searched using the function *std::list<S> getSolution().* When it is called, the algorithm searches for the next solution. If the next solution cannot be found, an exception of the *std::runtime_error* type is thrown with the message “No solution”.

An open list is a priority queue. For its implementation, we use the *std::multiset* class from the standard containers library*.* To order the vertices, a non-admissible cost estimate of the paths passing through them is used.

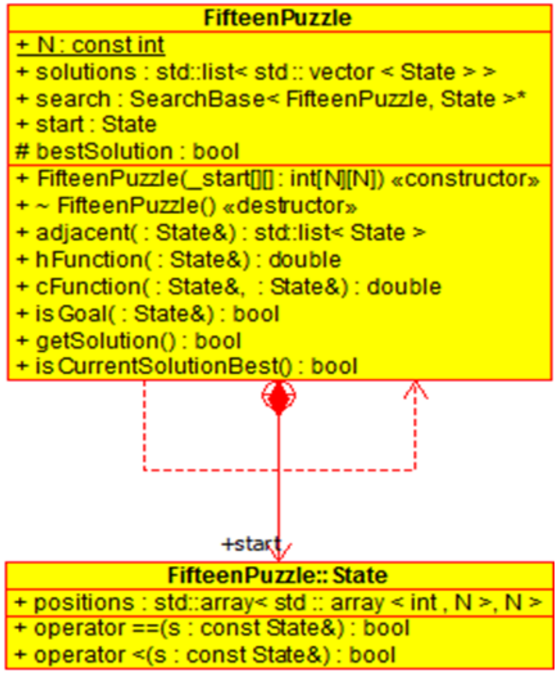
When implementing a closed list, special attention should be paid to the speed of checking that vertices are included in this list. The *std::multiset* class is also used to implement it, however, ordering is not based on the cost of the paths, but on using the “less” comparison operator.

Moscow, Russia
14-17 April, 2020

Implementation of Algorithms and Applications


G S
SearchBase
+ error : double
# graph : G*
+ SearchBase(graph : G*) «constructor»
+ ~ SearchBase() «destructor»
+ nextSolution() : std::list< S >
G S
RBFSRestarting
# start : Node*
# incumbent : Node*
# w : double
+ RBFSRestarting(graph : G*, start : undef, w : double) «constructor»
+ ~ RBFSRestarting() «destructor»
+ nextSolution() : std::list< S >
# RBFS(currentNode : Node*, limit : double, newLimit : double&) : Node*
G S
AStarRestarting
# open : std::multiset< Node >
# closed : std::multiset< Node, NodeComparator >
# start : Node*
# incumbent : Node*
# w : double
+ AStarRestarting(graph : G*, start : undef, w : double) «constructor»
+ ~ AStarRestarting() «destructor»
+ nextSolution() : std::list< S >
G S
AStarRepairing
+ weightDecreaseRate : double
# open : std::multiset< Node >
# closed : std::multiset< Node, NodeComparator >
# inconsistent : std::list< Node >
# start : Node*
# incumbent : Node*
# w : double
+ AStarRepairing(graph : G*, start : undef, w : double) «constructor»
+ ~ AStarRepairing() «destructor»
+ nextSolution() : std::list< S >
G S
AStarContinuing
# open : std::multiset< Node >
# closed : std::multiset< Node, NodeComparator >
# start : Node*
# incumbent : Node*
# w : double
+ AStarContinuing(graph : G*, start : undef, w : double) «constructor»
+ ~AStarContinuing() «destructor»
+ nextSolution() : std::list< S >

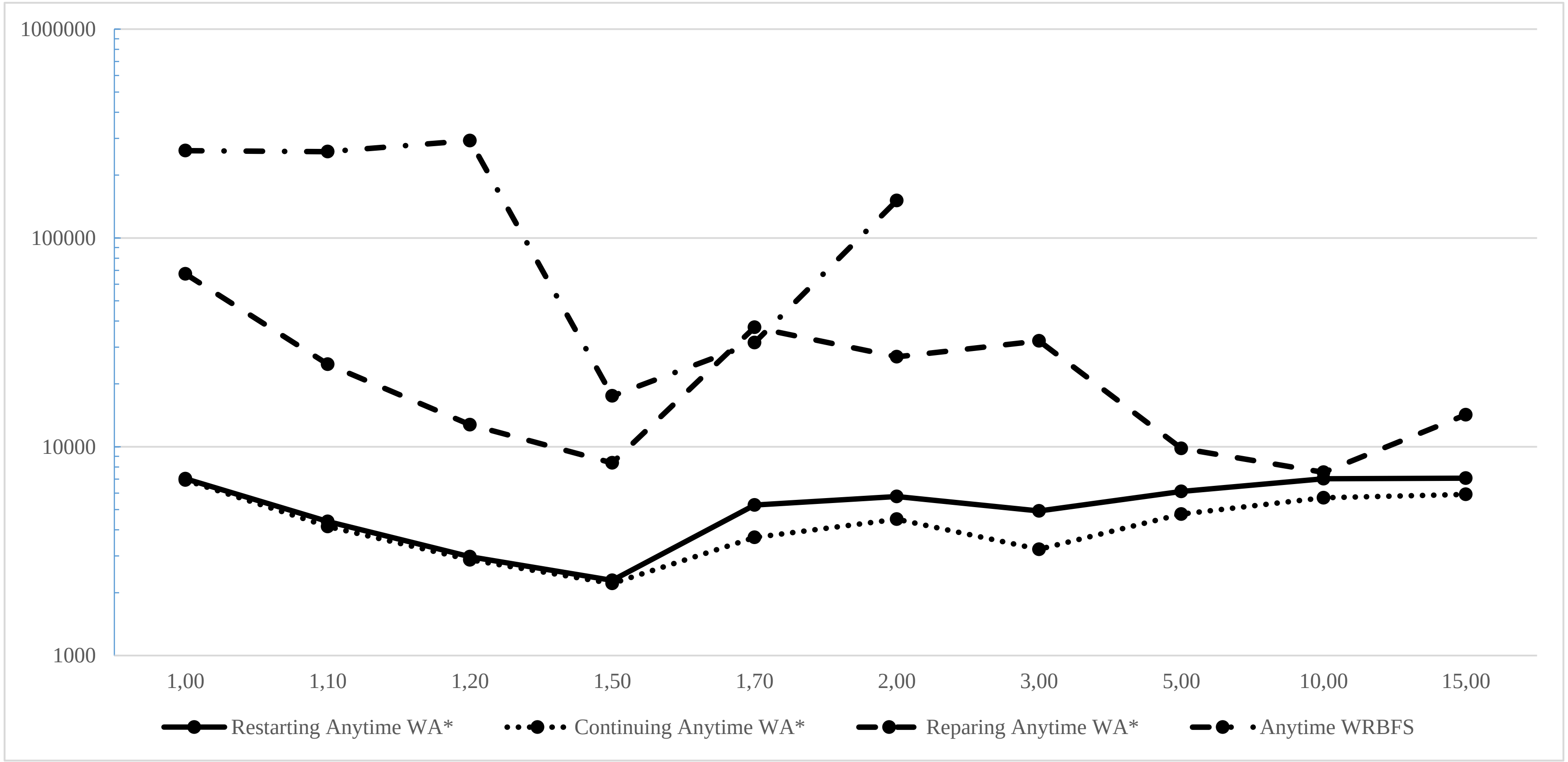
Implementation of Algorithms and Applications



```
FifteenPuzzle
+ N : const int
+ solutions : std::list< std :: vector < State > >
+ search : SearchBase< FifteenPuzzle, State >*
+ start : State
# bestSolution : bool
+ FifteenPuzzle(_start[][] : int[N][N]) «constructor»
+ ~ FifteenPuzzle() «destructor»
+ adjacent( : State&) : std::list< State >
+ hFunction( : State&) : double
+ cFunction( : State&,  : State&) : double
+ isGoal( : State&) : bool
+ getSolution() : bool
+ isCurrentSolutionBest() : bool
```

```
FifteenPuzzle:: State
+ positions : std::array< std :: array < int , N >, N >
+ operator ==(s : const State&) : bool
+ operator <(s : const State&) : bool
```

Algorithm Comparison



Conclusion

Our library of implementations of various anytime state-space search algorithms can be used to teach laboratory classes.

One of the use cases is to investigate the dependence of solution search times by algorithms at different parameters for an already implemented task (Eight puzzle). At the same time, it is possible both to compare the performance of one algorithm at different sets of parameters and to compare the performance of different algorithms between each other.

Another use case is to solve an arbitrary state space search problem using algorithms implemented in the library. At the same time, due to the universality of interfaces, it is possible to compare different algorithms on this task without unnecessary time to adapt the description of the task to different algorithms.

The first option is more suitable for cursory familiarization with the problem of searching in the space of states. The second option involves in-depth study, as it requires the student to implement the problem description by himself.

The algorithms and software tools focused on advanced of intelligent decision support systems are studied by students attending in the magistracy of the Department of Applied mathematics and Artificial Intelligence of National Research University “MPEI” in the direction “Applied mathematics and Informatics”.

Moscow, Russia
14-17 April, 2020

Thank you for attention!

Authors contacts:

Alexander Eremeev

eremeev@appmat.ru

Ilia Poliushkin
National Research University “MPEI”
PoliushkinIA@mpei.ru
www.appmat.ru

Nikolai Paniavin

PaniavinNA@mpei.ru