The Cheapest Way to Obtain Solution by Graph-Search Algorithms

Benedek Nagy

Eastern Mediterranean University, Faculty of Arts and Sciences,
Department Mathematics,
Famagusta, North Cyprus via Mersin 10, Turkey
E-mail: nbenedek.inf@gmail.com

Abstract: Graph-search algorithms belong to the set of basic problem-solving algorithms in Artificial Intelligence. There are systematic graphs search algorithms and also heuristic ones. Depending on the aim, e.g., to find any solution, all solutions, the best solution, one can choose an appropriate algorithm. The best-first algorithm is apt to find the best solution if a good heuristic is provided. Even, the obtained solution itself is the cheapest one, the way to obtain it may contain several useless branches. In this paper, a modified approach is shown which finds a solution having the minimal number of useless branches (depending also on the used heuristic). For the new algorithm, called minimum total cost search, the concept of the heuristic function is also changed: instead of predicting the cost of the closest goal state a kind of directed heuristic function is used: providing an estimation to the closest goal state from the given state to the given direction.

Keywords: Artificial Intelligence; problem solving; graph-search algorithms; cheapest way to obtain solution; minimum total cost search; best-first search; backtracking; heuristic search

1 Introduction

Some of the basic Artificial Intelligence algorithms are the backtracking and graph-search algorithms [1, 2, 7, 8]. Based on a state space (and the corresponding graph) representations, with their help one can find a solution of the modelled problem. Backtracking algorithm use minimal memory, actually, only the actual path is stored. At operator applications, a node with the newly obtained state is concatenated to the path. However, at backtrack steps the algorithm loses the information about the states (nodes) from that this step is made. Therefore, problems with graphs other than trees need special care. Graph-search algorithms provide other ways to obtain solutions. They store all states (nodes) that are already visited. Usually, they search/try several paths in parallel [5]. They explore all the states that can be obtained by an operator application from the state stored at the node by expanding that node. In this way, graph-search strategies do not enter to cycles and do not repeat to try paths proved to be dead end. This is one of their main advantages over backtracking.

Optimal search is a graph-search that provides optimal solution, i.e., path connecting the start node (initial state) with a goal state with a minimal weight. In a sense, this algorithm is very similar to the Dijkstra algorithm. Here, we want to recall the main difference between the two types of problems that are addressed by the Dijkstra algorithm and the optimal search. The Dijkstra algorithm is a very efficient shortest path search algorithm for graphs. In the problem, the graph is already given, and the algorithm, using dynamic programming technique, provides shortest path(s) starting from a given node. Opposite to this, in Artificial Intelligence, in the state space, the whole graph is not already given. The algorithm builds and explores those parts of the state space and its graph that are needed to find the optimal solution. The difference is more considerable when the graph cannot be discovered for free, but we need to pay for the discovering, i.e., for building the graph itself. In this paper, we consider the problem in this latter way. Our aim is not to find the cheapest path from the start to a goal state, but to find a solution and spend the least amount of cost for building the necessary part of the graph, i.e., the total cost of the whole construction (i.e. the search) is optimized.

2 Preliminaries

In this section we are describing the basic graph-search algorithms, and especially, the best-first search, but first we recall the concepts of state space and graph representation based on [7, 8]. For all non explained concepts the reader is referred to these standard textbooks on Artificial Intelligence.

2.1 The State Space

In Artificial Intelligence one of the most known methods to model problems is provided by state space which involves deterministic actions and complete information. The state space consists of four parts which is defined by 4-tuple (S,i,O,G). A set of states that is defined by S with a single initial state i, and a set of (target or) goal states G. O is a set of available operators (actions); for every $o \in O$ it is described for each state s whether the operator o is applicable by the function $o(s) = (s',c)$ that identifies the successor state s' of some state $s \notin G$ when action o is taken and also the cost c of applying o for s (where $o \in O$, $s \in S$). The value of $o(s)$ is empty in case o is not applicable on the state s.

There is a one to one correspondence between the definition of state space and its graph representation. The graph is defined by (N,E,v_0,T), where N is the set of vertices (nodes). Each state $s\in S$ is represented by a unique vertex, i.e., there is a bijection from S to N. The initial state i is represented by the start node v_0, where $v_0\in N$. The set of edges E contains an edge, i.e., $(v,v')\in E$ if and only if the vertices v and v' correspond to states s and s' and we have $o\in O$ such that $o(s) = (s',c)$ and the edge (v,v') have the label (o,c) indicating which operator with which cost is applied. The set of terminal nodes $T \subseteq N$ represents the goal states, i.e., there is a bijection between G and T. Since we have a bijection between S and N we may identify every vertex by a state, from here, we do not make difference between a vertex and the state it represents (if we do not have other ambiguity). Thus, we may simply refer to a vertex v by the state s assigned to it.

The aim is to find the/a path in the graph representation of the state space starting at the start node (representing the initial state) to a terminal node (representing a target state): the sequence $v_0o_1v_1o_2v_2\ldots v_{n-1}o_nv_n$ is such path if v_i represents state s_i and $o(s_i) = (s_{i+1},c_i)$ for some cost c_i and $s_n\in T$. The cost of a path is defined by $\sum_{i=1}^{n} c_i$, as usual.

In this paper we will use heuristic search methods, therefore we need the concept of heuristic function. It is usually defined in the following way:

A function which is an estimation of the distance of the state from the closest target state (if any) is called heuristic function, which assigns a nonnegative real value to each state $h : S \rightarrow \mathbb{R}^{\geq 0}$. It is assumed that $h(s) = 0$ if and only if $s\in T$.

See [3], a survey on heuristic functions in Artificial Intelligence, for more details.

2.2 Graph-Search Algorithms

In this subsection, we give the pseudo code of a general graph-search algorithm. These algorithms generate the graph representation of the problem, called search graph, dynamically. They use list data structure to represent nodes which includes: the (actual) state, the parent node, the operator was applied for generating the actual state, and the cost from the initial state to the actual state [8]. We note that if we do not have real costs, then the number of steps, i.e., the depth of the node is stored instead.

The data structure for a node is built up from

- a state s,
- pointer pt to the parent node,
- operator o that was applied to obtain s,
- cost c,
- heuristic value $h(s)$.

For non-heuristic search algorithms we do not need the heuristic values, that is we can write 1 (or the smallest positive unit used by the algorithm) for states not in *T* and 0 for states in *T*. This value will not modify the work of the algorithm.

The basic steps of graph-search algorithm are expanding the search graph as follows:

Algorithm 1 (Function Expand)

```
function Expand ($v$).
1. For each operator $o$ applicable to the state $s$ stored at node $v$ do
        1.1. Apply $o$ to $s$ and hence obtain ($s'$,$c$)
        1.2. If there is not vertex $v'$ such that it stores $s'$,
             Then create vertex $v'$ with data: $s',v,o,c_v+c$, where $c_v$ is
                 the cost stored at vertex $v$. Put vertex $v'$ into OPEN.
             Else decide whether or not to change the pointer pt of $v'$ to $v$
                 and its cost to $c_v+c$.
                 If $v'$ is found in the list CLOSED,
                 Then decide for each of its descendants in G whether or not
                     to rewrite the stored cost, accordingly.
2. End for
3. Return % End Expand
```

Algorithm 2 (Graph-search)

```
1. Create a search graph, G, containing only the start node, $s$:
2. Let the list OPEN contain only the initial state in the start node, $s$.
3. Create a list called CLOSED that is initially empty.
4. While (OPEN is non empty)
     4.1.  Select the first node in OPEN, remove it from OPEN, and put
           it on CLOSED. Call this node $v \in N$.
     4.2.  If $v \in T$, i.e., it is a target node,
           Then terminate successfully with the solution obtained by
               tracing a path along the pointers from $v$ to $s$ in G.
     4.3.  Call the function Expand with parameter $v$.
     4.4.  Reorder the list OPEN, %either according to some arbitrary
           scheme or according to heuristic merit%.
5. End while
6. Return failure (there is no solution with this representation).
```

The nodes in *OPEN* are the tip nodes of the graph-search, and the nodes on *CLOSED* are the non-tip nodes. In Breadth-first and Depth-first search algorithms there is no real cost function, unit cost, e.g., the depth is used. In the Breadth-first search algorithm with a queuing function the newly generated states put at the end of the queue, after all the previously generated states. However, the Depth-first search algorithm puts the newly generated states at the front of the queue (more like a pushdown stack architecture).

The algorithm ends successfully whenever the selected node for expansion is a target node. The desirable path $v_0o_1v_1o_2v_2\ldots v_{n-1}o_nv_n$ can be obtained by tracing the pointers from v_n to v_0 in the reverse order (if the/a solution exists). Whenever the set of terminal nodes are empty, i.e., $T = \emptyset$, or no terminal node can be obtained from the start node using the given operators, the algorithm terminates reporting unsuccessful search (failure).

Since this is a graph-search algorithm some of the states obtained by Expand may already be in *OPEN* or *CLOSED*, i.e., they have already been generated. Recognizing and deciding what to do if some of the newly generated data are already generated before, has some computational cost. In the simplest cases, e.g., when the cost of the obtained solution is not of high importance, there is no extra process for the states that are already in the database. For Breadth-first, Depth-first, and, as we will see, for best-first search algorithms in the else branch of step 1.2 of the function Expand can be deleted (no change will happen if a state is already stored in some nodes of the database, i.e., in the already obtained search graph).

In this paper, we would like to obtain a solution as fast and in the cheapest way as possible. Therefore, two specific graph-search algorithms, namely, the optimal search and the best-first search are the most important for us. We briefly recall them in the next two subsections.

We note that relations of graph-search and parallel algorithms are investigated in [1, 5]. However, in this paper, we mainly deal with the traditional, sequential approach.

2.3 The Optimal Search

A systematic search algorithm that provides the optimal (minimal cost) solution (in case a solution exists) is the optimal search. Since the aim is to find the best solution (minimal cost path from the start node to a terminal node) in the Expand function if a cheaper path is found to a node v' than the actually stored path (through the links to the parent nodes), then the parent node of v' and the cost of the path to node v' are updated. However, it is not possible to find a cheaper path to node that is already in the list *CLOSED*. In this search algorithm, the list *OPEN* is sorted in a non decreasing way by the stored cost of the path to the node.

At the next algorithm, we do not want to get the optimal solution (for sure), but we want to obtain a solution as fast as possible. This can be done by the use of a heuristic function.

2.4 The Best-First Search

One of the search algorithms that searches in a graph by selecting and expanding the most promising node based on the special rule is the Best-First search. This algorithm selects the promising node by using a heuristic function, $h(s)$, which takes a state (stored in a node v) as argument and returns a non-negative real number. At this algorithm, the nodes in the list *OPEN* are ordered (may be based on the variety of heuristics ideas) so that the "best" one will be the first in the list, and thus, it can be easily selected. The heuristic function estimates the cost of a path from node v to a target node. We then expand the node to generate its successors and quit if one of them is a target. Otherwise, among all the other nodes (including the newly generated ones), again the most promising node (i.e., the node containing the state with the smallest heuristic value among the nodes stored in the list *OPEN*) is chosen and the process continues. By using a heuristic function, the search can be informed about the direction to a target and predict how close the end of a path is to a target [6, 7]. It is known that Best-First search can provide a solution in the fastest way if the heuristic function is informative enough.

The function Expand for best-first search algorithm is specified from Algorithm 1, as we have already indicated, by simply deleting the Else branch of step 1.2. In Algorithm 2 the reordering of the list *OPEN* (step 4.4) goes by non-decreasing values by the heuristic values of the states stored.

We note that in [4] search algorithm based on the best-first is presented using the memory in an efficient way.

3 The New Approach

The new approach is somewhat similar to the Best-First search algorithm, it is a kind of modification of the usual graph-search algorithms reducing the cost of the applied operators. The heuristic function is also redesigned for this purpose.

3.1 Appropriate Type of Heuristic Functions

The heuristic function, to obtain our aim, has two arguments, not only a state, but also a chosen direction to go forward from that state, i.e., an operator. By the help of this new feature, we can reduce the cost of the applied operations. In this new algorithm, the data structure that is stored contains vertices with the following data structure:

- a state *s*,
- pointer *pt* to the parent node,
- operator *o* that was applied to obtain *s*,
- cost *c*,
- an applicable operator *o*' (that is chosen to apply to state *s*)
- heuristic value *h*(*s*,*o*').

The heuristic function is applied to each element of the *OPEN* list. In this algorithm, we have again the two lists. At a node the state, pointer, operator and cost are the same as before. The new part is a chosen operator and the new type of heuristic function. Now, we are ready to present the new algorithm with the appropriately modified Expand function.

3.2 The Minimum Total Cost Search

To reduce the cost (of the applied operations) in the Expand function, we should not use all the applicable operations for a given state at a time, we continue the search only in a chosen direction. In this way, the algorithm has a similarity to the backtracking search. The list *OPEN* is assumed to be sorted by the stored heuristic value *h*(*s'*,*o*) in non-decreasing order.

Algorithm 3. (Function Best-Continue)

```
function Best-Continue(v)
1.  Apply o' stored in v to s stored in v and hence obtain (s',c).
2.  If there is no such node in OPEN and CLOSE that stores the state s'
    Then
    2.1. If s' is a goal state,
         Then terminate with the solution reaching s' from s
              (and its predecessors).
    2.2. For each operator o applicable to the state s' do
              Create a node v' to store state s', pt to node v, o', c, then the
              applicable operator o and the heuristic value h(s',o).
              Insert v' into the list OPEN in non-decreasing way by h(s',o).
    2.3.  End for
3.  Return % End Best-Continue
```

Algorithm 4. (Minimum Total Cost Search, MTCS)

```
1. Create a search graph, G, containing only the initial state with its
   applicable operators:
2. For each operator o applicable to the initial/start state s do
   2.1. Let the list OPEN contain the node: s. NUL, NUL, 0, o, h(s,o).
        Insert it into OPEN by the values h(s,o) in non-decreasing way.
3. End for
4. Create a list CLOSED that is initially empty.
5. While (OPEN is non empty)
   5.1.   Select the first node of OPEN, remove it from OPEN, and put
          it into CLOSED. Call this node v∈N.
   5.2.   Call the function Best-Continue with parameter v.
6. End while
7. Return failure (there is no solution with this representation).
```

The algorithm is complete and correct: For finite search graphs, if a solution exists, it will find a solution; and it terminates with failure if there is no solution. However, for problems represented by infinite graphs, it may not produce solution if, e.g., the heuristic function directs the search to an infinite branch without solution. This is actually, the same phenomenon for other heuristic search algorithms, including heuristic backtracking and best-first. (To overcome this issue one may use the optimal search, or its mix with the best-first, i.e., the A- or A*-algorithms.) However, if the heuristic is not completely misleading in an infinite search space, the new algorithm can be applied. Moreover, due to the new type of heuristic function, the total cost of the explored search tree is minimized (according to the quality of the heuristic function).

The new approach is between the graph-search and backtracking, by breaking the expand function into parts applying operators one-by-one, similarly as they are used in backtracking algorithms. On the other side, since we have no backtrack operation, we keep all the already explored parts of the search graph, the new method also inherits the advantages of graph-search algorithms (e.g., loops and alternative paths to the same dead end do not cause such problems that could occur at backtracking).

We note that by efficient practical design we do not really need to store the repeated information (i.e., *s*, *pt*, *o* and *c*) at nodes storing the same state. By using linked (multi)lists we can store this part of the information separately and this part can be linked by a pointer to the real node that contains additionally *o'* and *h*(*s*,*o'*).

3 Example

In this section, an example is shown. We also compare the work of the heuristic backtracking, the optimal search, the best-first and our new MTCS algorithm. Let the graph representation of our problem be given as it is shown in Figure 1. The cost assigned to the operator application is written at the left side of the edges by blue color. The heuristic values (indicated by red) are given in Table 1.

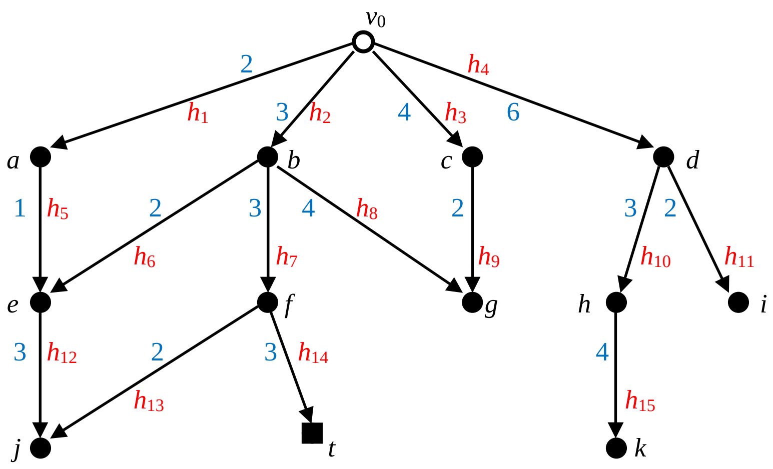

Figure 1
The graph of the state space representation of the example

In the example, we use two heuristic functions, the perfect heuristic and a non-perfect (another) heuristic (which is more realistic), as specified in Table 1.

Table 1
Heuristic values used in the example

h_i	$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
perfect		99	9	99	99	99	99	6	99	99	99	99	99	99	3	99
another		5	10	11	14	4	5	6	4	7	6	8	3	4	3	4

$h(v)$	v:	v_0	a	b	c	d	e	f	g	h	I	j	t	k
perfect		9	99	6	99	99	99	3	99	99	99	99	0	99
another		10	4	5	7	7	3	3.5	99	4	99	99	0	99

The upper part of the table shows the new type of heuristic functions that are apt to use for backtracking and the MTCS algorithm. The lower part of the table shows traditional heuristic functions that can be used for the best-first algorithm. For the "another" heuristic the average new-type heuristic values of the out-edges are used at the corresponding nodes.

Let us see, how the solution is obtained by various algorithms. Let us start with the optimal search. Obviously, it starts by expanding the node v_0. By applying all the 4 operators it obtains 4 new nodes (*a*,*b*,*c*,*d*) and this costs 2+3+4+6=15. The nodes in *OPEN* are ordered by their cost value, and thus, node *a* is expanded in the next step with cost 1: node *e* is obtained. In the next 2 steps nodes *b* and *e* are expanded, by costs 9 and 3, respectively; adding the nodes *f*, *g* and *j* to the explored search graph. Then, node *c* is expanded having a better path to *g*, and adding 2 to the total cost (of the graph we have paid so far). Now the nodes *j*, *f*, *g* and *d* are expanded: this gives 0+5+0+5 cost and adding nodes *t*, *h* and *i* to the explored part of the graph. Node *i* has the smallest cost among the nodes in *OPEN*, therefore it is expanded (it is moved to *CLOSED* without other changes). Now, *t* is among the nodes with the smallest cost in *OPEN,* and it can be tested that it contains a goal state. The solution is found: v_0 to *b*, then to *f* and from *f* to *t*. Even the cost of the solution is 9, the cost of the explored part of the search graph is 40.

Now let us see how the heuristic search algorithms works with perfect heuristic. The best first must start by expanding the start node v_0 and, hence, obtaining nodes *a*, *b*, *c*, and *d*, it costs 15. Then, the smallest heuristic value is 6 and it is at node *b*. By expanding *b* the nodes *e*, *f* and *g* are obtained with additional cost 9. Now, the smallest heuristic value is 3 (at *f*) among the nodes in *OPEN*. Consequently, expanding *f*, the nodes *j* and *t* are explored (cost 5), and the node *t* has 0 heuristic value. The search is finished, the solution is found, and the total cost of the search was 29.

The backtracking using the perfect heuristic given in the first row of the table, will go from v_0 to *b* and, then to *f*, and it finds the terminal node *t* in the next step. The total cost of the search is 9, it is exactly the same as the cost of the solution (it was obtained without backtrack steps).

Algorithm MTCS works in a similar manner. In the beginning, it has basically 4 edges in the open list indicated by the values h_1, h_2, h_3 and h_4. It choses the edge v_0 to *b* as h_2 is the smallest among the values in the list *OPEN*. Using Best-continue, the edges indicated by h_6, h_7 and h_8 are found and stored in *OPEN*. Then, the edge with h_7 is chosen for Best-continue. In the next round, by the edge with h_{14} the solution is found. The total cost of the MTCS search was 9, same as the total cost of the backtracking (with perfect heuristic values).

Let us see the performance of the heuristic search algorithms with a non-perfect heuristic function. One can easily check that, in our example, the best-first search finds the solution by using the heuristic values from the last row of Table 1, with an even larger total cost than previously: it is 33. It is usual that without a really good heuristic function the performance of the heuristic search algorithms are worst than with them. (Actually, the quality of the heuristic function is essential.)

Let us see, how the backtracking works: It will go from v_0 to *a*, and then to *e*, and to *j* (with total cost 6, so far). In the next step, it realizes the dead end, and backtracks to v_0. Then, it goes to *b* and then to *g*. The total cost of used operators is 6+7=13, so far. However, it realizes the dead end, and backtracks to *b*. Then it continues to *e*, again, moreover to *j* (the total cost is 13+5=18, so far). Again, realizing the dead end backtracks and backtracks to *b*. Now, it goes to *f*, and, finally, from *f* it finds the terminal node *t*. The total cost of the search is 24, it is much larger than the cost of the solution, even the backtrack steps had no direct extra costs.

Finally, we show the performance of MTCS with the non-perfect heuristic values specified in Table 1. Again, the list *OPEN* contains the edges indicated by the values h_1, h_2, h_3 and h_4. However, in this case, h_1 looks the best choice (and it costs 2). Then instead of that edge the edge with h_5 will be in the list *OPEN*. In the next round, it seems the best choice and the edge indicated by h_{12} will replace it in *OPEN*. Then edge with h_{12} is chosen and moved to *CLOSED*. The total cost of the search is 6, so far. In the next round the edge h_2 is chosen for Best-continue. Consequently, *OPEN* will include the edges with h_6, h_7 and h_8. Then the edges with h_8 and h_6 are chosen, and both become *CLOSED*. The total cost is 6+9=15 up to this point. Then the choice of the edge with h_7 to Best-continue gives two new *OPEN* edges: the ones with h_{13} and h_{14}. The latter one is a better choice and the algorithm terminates with the solution. The total cost to find it was 21, less than the costs of the other algorithms. Of course, without perfect heuristic values we usually need to pay some extra costs to explore some parts that are not directly needed for the solution, however, by our algorithm this extra cost is minimized.

Backtracking algorithms keep only a path in their memory, and thus, with a heuristic backtracking algorithm with a good heuristic function the solution may be obtained without any backtrack steps (see, the solution with perfect heuristic). In this way, the total cost to find the solution is exactly the same as the cost of the solution itself. However, if the heuristic function is not good enough, the total cost to obtain the solution increases, and disadvantages of the backtracking algorithm may occur, e.g., by applying (and paying again for) the same operator at the same state reached in a newer path, as we have seen at the case of “another” heuristic.

Even best-first could be believed as a method exploring the minimal part of the search graph to obtain a solution, we have shown that our new algorithm can work with even less cost. The best-first pays extra fees when at expand, it is applying operator(s) at a node not only in the ideal direction.

Conclusions

There are some cases in real life when we do not have a chance to freely see what and how will happen if we do something (modelled by applying an operator). In these cases, to try and analyze how a given step may help to find a solution (to reach a goal state), one may need to pay the cost of this step (operator). Consequently, the problem to find the cheapest (optimal) solution shifts to the problem to obtain a solution in the cheapest way, i.e., exploring a minimum-cost part of the search tree that is needed for the solution.

In this paper, we have modified the well-known search algorithms and we have obtained a general heuristic algorithm to have a search algorithm with minimum total cost. The algorithm is related to the best-first graph-search algorithm and also to heuristic backtracking algorithms uniting their advantages for the considered types of problems.

References

[1] Kenneth A. Berman, Jerome L. Paul: Algorithms: Sequential, Parallel, and Distributed. Thomson/Course Technology, 2005

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms. MIT Press and McGraw-Hill, 1990 (3rd edition, 2009)

[3] Fred Glover, Harvey J. Greenberg: New Approaches for Heuristic Search: A Bilateral Linkage with Artificial Intelligence, European Journal of Operational Research 39/2 (1989) 119-130

[4] Richard E. Korf: Linear-Space Best-First Search, Artificial Intelligence 62/1 (1993) 41-78

[5] Benedek Nagy: On the Notion of Parallelism in Artificial and Computational Intelligence, Proceedings of the 7th International Symposium of Hungarian Researchers on Computational Intelligence, Budapest, Hungary (2006) pp. 533-541

[6] Ira Pohl: Heuristic Search Viewed as Path Finding in a Graph, Artificial Intelligence 1/3-4 (1970) 193-204

[7] Elaine Rich, Kevin Knight, Sivashankar B Nair: Artificial Intelligence. Tata McGraw-Hill, 3rd edition, 2008 (Elaine Rich, Kevin Knight: Artificial Intelligence, 1st edition, 1983)

[8] Stuart Russell, Peter Norvig: Artificial Intelligence – A Modern Approach. Prentice Hall, 1995 (3rd edition, 2009)