Journal of Theoretical and Applied Computer Science
ISSN 2299-2634 (printed), 2300-5653 (online)
Vol. 8, No. 2, 2014, pp. 63–71
http://www.jtacs.org

The efficiency of the A* algorithm's implementations in selected programming languages

Jacek Klimaszewski

Faculty of Computer Science and Information Technology, West Pomeranian University of Technology, Szczecin, Poland

jacek.szachista@gmail.com

Abstract: *This article presents results of the experiment, in which data structures, used by the A* algorithm (i.e. priority queue and hash table), were tested. A* algorithm was used to solve 15-puzzle, where puzzle's state was kept in the 64-bit integer variable. The algorithm used either built-in data structures (such as Python's dictionary) or provided by a standard library (such as unordered_map in the C++ Standard Template Library).*

Keywords: *path finding, data structures, artificial intelligence*

1. Introduction

A* algorithm [1] is widely used in graph traversal problems (e.g. pathfinding) because of its performance and accuracy (much better than Dijkstra's algorithm). It can also be used to find solution of some logic games (puzzles, sudoku etc.). Here it was used to solve 15-puzzle.

The new standard of C++ language (called C++11) introduced new constructions (e.g. lambda expressions, *r-value* reference) and extended standard library by adding new containers, threading library and many more [2]. The author focused on *unordered_set* class, because it could be used to implement *closed set* efficiently. C++ implementation of an A* algorithm was compared against implementations in Python [3] and Java [4].

As mentioned in the abstract, *bitmap representation* was used to held puzzle's state in the memory because of 4 reasons:

1. it occupies less memory than a straightforward array representation,
2. it fits into processor's registry (on 64-bit architecture),
3. it can be used as a key to the hash table,
4. operations on single variable should be faster than operations on array.

In Section 2, 15-puzzle and its representation in a computer memory are described. Section 3 describes A* algorithm and its implementation. In the next section results of the experiment are presented and discussed. At the end of the article there is source code of the C++ program used in benchmark to solve 15-puzzle.

2. The *15-puzzle* problem

This chapter brings some details about what *15-puzzle* is and how to keep its position efficiently in the memory.

2.1. History

The "15-puzzle" is a sliding square puzzle commonly (but incorrectly) attributed to Sam Loyd. However, research by Slocum and Sonneveld (2006) has revealed that Sam Loyd did not invent the 15-puzzle and had nothing to do with promoting or popularizing it. The puzzle craze that was created by the 15-puzzle began in January 1880 in the United States and in April in Europe and ended by July 1880. Loyd first claimed in 1891 that he invented the puzzle, and he continued until his death a 20 year campaign to falsely take credit for the puzzle. The actual inventor was Noyes Chapman, the Postmaster of Canastota, New York, and he applied for a patent in March 1880.

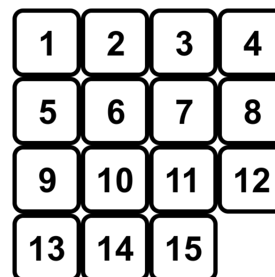


Figure 1. A solved 15-puzzle.

The 15-puzzle consists of 15 squares numbered from 1 to 15 that are placed in a 4×4 box leaving one position out of the 16 empty. The goal is to reposition the squares from a given arbitrary starting arrangement by sliding them one at a time into the configuration shown on the Figure 1. For some initial arrangements, this rearrangement is possible, but for others, it is not [5].

2.2. Problem overview

Solving this puzzle is a typical graph traversal problem — from a start node find the shortest path (if exists) to the goal node. There are many algorithms, that can be used to solve this problem[1]. Author chose A* algorithm to test 2 data structures — priority queue (it is needed to select best node to expand, based on estimated cost) and a hash table (to check quickly whether selected node was visited or not).

2.3. Implementation notes

There are 16 squares (15 + one empty), so 4 bits are enough to number every tile ($2^4 = 16$), therefore whole position can be mapped to a 64-bit number ($16 \cdot 4 = 64$), which will be used as a key to the hash table. By assumption the least significant nibble describes left top square while the most significant nibble represents right bottom square (which is empty in the solved puzzle state). As a result, a solved puzzle state describes number $\mathrm{FEDCBA9876543210}_{16}$ and the puzzle on the Figure 2 is represented by the number $\mathrm{5DB86E2AC3F49170}_{16}$, where `0` stands for a tile **1**, `1` stands for a tile **2**, ..., and F_{16} (15) stands for an empty square.

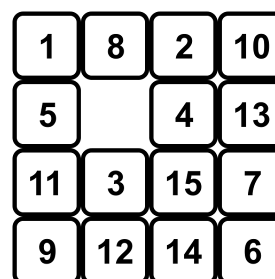


Figure 2. A shuffled 15-puzzle.

The first (left top) square has index 0, the next one has index 1 and so on from the left to the right and from the top to the bottom (the last square has index 15). The operation of division by 4 gives the number of the row, while remainder of this division produces index of the column. To determine what tile is in the n-th square, the number (in binary system) must be shifted right by $4 \cdot n$ bits (in hexadecimal system it is equivalent for division by 16^n) and

[1] e.g. Depth-first search, Breadth-first search, B*, ...

all but 4 lower bits must be ignored (it is equivalent for **mod 16** operation and in hexadecimal system is equal to the last digit). For example:

$$\begin{aligned} s &= \mathrm{5DB86E2AC3F49170}_{16} = 6753268771197325680_{10} \\ s \ \mathbf{div}\ 16^5 &= \mathrm{5DB86E2AC3F}_{16} = 6440418978879_{10} \\ \left(s \ \mathbf{div}\ 16^5\right) \ \mathbf{mod}\ 16 &= \mathrm{F}_{16} = 15_{10} \end{aligned}$$

So the 5th (actually 6th) square is an empty square.

It is a good idea to keep an index of the empty square to find possible moves more quickly. Below there is part of the program which moves tile upward (if possible) to the position of the empty square.

```
1  struct State
2  {
3     unsigned long s;   // 64-bit integer number without sign
4     int n;             // index of the empty square
5     int f;             // sum of g and h
6     int g;             // actual cost from the beginning to the current state
7     int h;             // estimated cost form the current state to the goal
8  };
9
10 State move_up(State state)
11 {
12    if (state.n / 4 < 3)
13    {
14       int n = state.n + 4;
15       unsigned long m = 0xfUL << (n * 4),
16             v = (state.s & m) >> (n * 4),
17             s = state.s & ~(m | (0xfUL << (n * 4)));
18       state.s = s | (v << (state.n * 4)) | (0xfUL << (n * 4));
19       state.n = n;
20    }
21    return state;
22 }
23
```

In the 9th line program checks whether the empty square is not in the last row. If so then a new position of the empty square is calculated. In the next 3 lines mask is computed (`0xfUL` stands for unsigned long hexadecimal constant), number of the tile to be moved is extracted and these squares are removed (by masking) from the state. Finally they are exchanged and put in the new state in the line 15th. Also new position of the empty square is changed. At last results of these operations are returned in the line 18th. The other 3 functions (move_down, move_left, move_right) differ from each other only by lines 9th and 11th respectively.

For example: assume that `state = {0x5DB86E2AC3F49170UL, 5}`. Condition in the 9th line is satisfied, because $5 \ \mathbf{div}\ 4 = 1 < 3$. A new position of the empty square is $n = 9$. Below you can see, how these operations in the next few lines are performed in binary system (*M* stands for value computed in the 14th line, behind the `&` operator).

$$
\begin{aligned}
state.s &= 0101110110111000011011100010101011000011111101001001000101110000_2 \\
m &= 0000000000000000000000001110000000000000000000000000000000000000_2 \\
M &= 1111111111111111111111110000111111111111000011111111111111111111_2 \\
state.s \,\&\, m &= 0000000000000000000000000010000000000000000000000000000000000000_2 \\
s = state.s \,\&\, M &= 0101110110111000011011100000101011000011000001001001000101110000_2
\end{aligned}
$$

3. The A* algorithm

A* algorithm is a kind of balance between Dijkstra's algorithm and Best-first search algorithm. The cost function $\hat{f}(n)$ in the A* algorithm is a sum of 2 functions: $\hat{g}(n)$ (which represents actual cost from the start node to the current node n) and $\hat{h}(n)$ (which estimates cost from the current node n to the goal node). The $\hat{h}(n)$ function must be admissible, i.e. it must not overestimate true cost to the goal node. When $\hat{h}(n) = 0$, A* algorithm behaves like Dijkstra's algorithm.

3.1. Pseudocode

Each node can remember its predecessor and values of $\hat{f}$, $\hat{g}$, $\hat{h}$ functions.

Algorithm 1 A* algorithm pseudocode

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL do
    current = remove lowest rank item from OPEN
    add current to CLOSED
    for neighbours of current do
        cost = g(current) + movementcost(current, neighbour)
        if neighbour in OPEN and cost less than g(neighbour) then
            remove neighbour from OPEN, because new path is better
        end if
        if neighbour in CLOSED and cost less than g(neighbour) then
            remove neighbour from CLOSED
        end if
        if neighbour not in OPEN and neighbour not in CLOSED then
            set g(neighbour) to cost
            add neighbour to OPEN
            set priority queue rank to g(neighbour) + h(neighbour)
            set neighbour's parent to current
        end if
    end for
end while
reconstruct reverse path from goal to start by following parent pointers
```

During execution, A* algorithm operates on 2 sets: the first contains nodes to be expanded (open set), and the second contains already visited nodes (closed set). Priority queue is used as an open set (nodes with lower value of the cost function should be expanded first), while hash table is usually used as a closed set (operations of insertion and searching have $\mathcal{O}(1)$ time complexity in optimistic case). Moreover, priority queue is usually implemented as a binary heap, hence operations of insertion and removal have $\mathcal{O}(\log n)$ time complexity in average case, and access to the element with the highest priority is constant in time (n is a number of elements in a binary heap).

Operation of checking if element belongs to priority queue has $\mathcal{O}(n)$ time complexity. To avoid searching when element neither is in the priority queue nor it may be improved, additional hash table was used.

Elements in a C++ priority queue are ordered descending, therefore sign of the $\hat{f}$ function was inverted. Furthermore, all priority queues assume, that elements are immutable, but in C++ and Python it was possible to add functionality of updating an element without writing own implementation of a priority queue. In Java it was not possible, until own version of a priority queue was written.

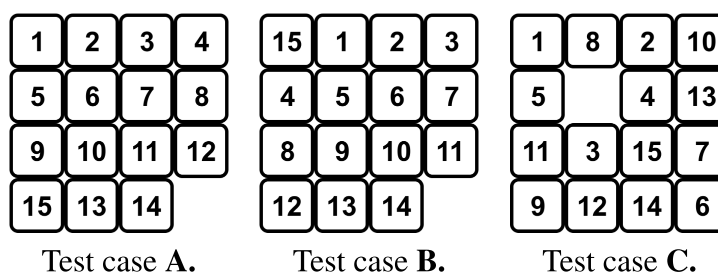
3.2. Manhattan distance

To estimate cost between current node and a goal node (i.e. value of a $\hat{h}(n)$ function), Manhattan distance is used — for every tile (even blank tile) it is checked, how its position is far from the initial position (e.g. the distance of the tile **8** from the initial position on the Figure 2 is 3 — 1 row + 2 columns), and all these distances are summed up.

4. Results

Program solving 15-puzzle was written in 3 languages: C++, Java and Python. All programs were tested on the same machine — *Samsung NP-RC520-S06PL* with 8 GB RAM and with operating system *Ubuntu 13.04 x64*. C++ program was compiled in *g++* 4.7.3 using command `g++ -O2 -std=c++0x prog.cpp`, Python script was launched in Python 2.7.4 interpreter, and version of Java was 1.7.0_51.

To compare performance of programs, three test cases were used: easy (**A**), medium (**B**) and sophisticated (**C**).



Test case **A.** Test case **B.** Test case **C.**

Table 1 contains result of the experiment, in which time of execution, number of visited nodes (nodes in closed set) and number of active nodes (nodes in open set) were measured. The solution of each test case (first found) is shown in separate row.

As we can see, number of active nodes in C++ and Python program is the same, because implementations of priority queue in these languages are probably the same. The difference between C++ and Java is a result of a different implementation — in Java, if node in priority queue could be improved, it was deleted and then better node was inserted. In each test, C++

Table 1. Results of tests.

<table>
<tr><th>Case</th><th>Language</th><th>Time (s)</th><th>#visited nodes</th><th>#active nodes</th></tr>
<tr><td rowspan="3">A.</td><td>C++</td><td>0.00</td><td>948</td><td>895</td></tr>
<tr><td>Java</td><td>0.05</td><td>1012</td><td>942</td></tr>
<tr><td>Python</td><td>0.08</td><td>948</td><td>895</td></tr>
<tr><td colspan="5">→→→↓←↑←←↓→→→↑←←↓←↑</td></tr>
<tr><td rowspan="3">B.</td><td>C++</td><td>9.08</td><td>671916</td><td>575583</td></tr>
<tr><td>Java</td><td>267</td><td>805060</td><td>691479</td></tr>
<tr><td>Python</td><td>2430</td><td>671916</td><td>575583</td></tr>
<tr><td colspan="5">→→↓↓→↓←←↑→↑→↑←←←↓↓→↑→→↓←←↓←↑↑→→→↑←←↓→↓←←↑↑→↓←↑</td></tr>
<tr><td rowspan="3">C.</td><td>C++</td><td>17.24</td><td>872092</td><td>736763</td></tr>
<tr><td>Java</td><td>326</td><td>775886</td><td>651863</td></tr>
<tr><td>Python</td><td>11850</td><td>872092</td><td>736763</td></tr>
<tr><td colspan="5">↓←↑←↑→→↓←↑↑←↓↓↓→↑↑→↓←←↑→↑→↓→↑←↓↓←↑↑→↓↓←←↑→↑←</td></tr>
</table>

program was the fastest — the next was Java and the last was Python. Surprising is fact, that Python program was much more slower than Java — probably representation of a puzzle's state was not efficient enough in this language.

5. Summary

The intention of this experiment was to use everything, what offers standard library in selected programming languages, to implement A* algorithm. As expected, C++ overtook Java and Python. It also showed, that bitmap representation is not efficient in scripting languages. It is worth checking whether better implementation of a priority queue (incorporating hash table which holds indexes of elements in a priority queue's array) can improve performance of a program.

References

[1] Peter E. Hart, B. R., Nils J. Nilsson: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE transactions of systems science and cybernetics, 4(2), 1967.

[2] Stroustrup, B.: What is C++0x? CVu, 21, 2009.

[3] Lambert, K. A.: Fundamentals of Python: First Programs. Cengage Learning, 2011.

[4] Eckel, B.: Thinking in Java: The Definitive Introduction to Object-Oriented Programming in the Language of the World-Wide Web, 3rd Edition. Prentice Hall PTR, 2002.

[5] Slocum, J., Weisstein, E. W.: 15 Puzzle. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/15Puzzle.html. Last visited on 20/04/2014.

Appendix

Below you can find C++ source code of the program used in testing. Bitwise operations (shifting and masking) were used instead of division and modulo operations in all 3 programs.

```
1  #include <iostream>
2  #include <iomanip>
3  #include <string>
4  #include <queue>
5  #include <vector>
6  #include <unordered_set>
7  #include <cstdlib>
8  using namespace std;
9
```

```
10  class State_16
11  {
12      friend ostream& operator << (ostream& out, const State_16& s);
13  private:
14      unsigned long s;
15      int n, f, g, h;
16      const State_16 *parent;
17  public:
18      State_16() : s(0xfedcba9876543210UL), n(15), f(0), g(0), h(0), parent(nullptr) {}
19
20      State_16(unsigned long s, int n, int g) : s(s), n(n), f(0), g(g), h(0), parent(nullptr)
21      {
22          int i, j, k;
23          unsigned long m;
24          for (i = 0, j = 0, m = 0xfUL; m != 0; i++, j += 4, m <<= 4)
25          {
26              k = ((s & m) >> j);
27              h += abs((k >> 2) - (i >> 2)) + abs((k & 3) - (i & 3));
28          }
29          f = -g - h;
30      }
31
32      State_16 move_up() const
33      {
34          if ((n >> 2) < 3)
35          {
36              int N = n + 4;
37              unsigned long m = 0xfUL << (N << 2), v = (s & m) >> (N << 2), S = s & ~(m | (0xfUL << (n << 2)));
38              return State_16(S | (v << (n << 2) | (0xfUL << (N << 2))), N, g + 1);
39          }
40          return *this;
41      }
42
43      State_16 move_down() const
44      {
45          if ((n >> 2) > 0)
46          {
47              int N = n - 4;
48              unsigned long m = 0xfUL << (N << 2), v = (s & m) >> (N << 2), S = s & ~(m | (0xfUL << (n << 2)));
49              return State_16(S | (v << (n << 2) | (0xfUL << (N << 2))), N, g + 1);
50          }
51          return *this;
52      }
53
54      State_16 move_left() const
55      {
56          if ((n & 3) < 3)
57          {
58              int N = n + 1;
59              unsigned long m = 0xfUL << (N << 2), v = (s & m) >> (N << 2), S = s & ~(m | (0xfUL << (n << 2)));
60              return State_16(S | (v << (n << 2) | (0xfUL << (N << 2))), N, g + 1);
61          }
62          return *this;
63      }
64
65      State_16 move_right() const
66      {
67          if ((n & 3) > 0)
68          {
69              int N = n - 1;
70              unsigned long m = 0xfUL << (N << 2), v = (s & m) >> (N << 2), S = s & ~(m | (0xfUL << (n << 2)));
71              return State_16(S | (v << (n << 2) | (0xfUL << (N << 2))), N, g + 1);
72          }
73          return *this;
74      }
75
76      vector<State_16> children() const
77      {
78          vector<State_16> children;
79          State_16 state = move_up();
80          if (state.s != s)
81              children.push_back(state);
82          state = move_down();
83          if (state.s != s)
84              children.push_back(state);
85          state = move_left();
86          if (state.s != s)
87              children.push_back(state);
88          state = move_right();
89          if (state.s != s)
90              children.push_back(state);
91          return children;
92      }
93
94      bool is_terminate() const {return s == 0xfedcba9876543210UL;}
95      void set_parent(const State_16& parent) {this->parent = &parent;}
96      unsigned long key() const {return s;}
97      int get_f() const {return f;}
98      int get_g() const {return g;}
99      bool operator < (const State_16& state) const {return f < state.f;}
100     bool operator > (const State_16& state) const {return f > state.f;}
101     bool operator == (const State_16& state) const {return s == state.s;}
102
103     static const State_16 BAD;
104
105     static void trace(const State_16& terminal)
```

```
106         {
107             const State_16 *s = terminal.parent;
108             State_16 current = terminal;
109             string path;
110             while (s != nullptr)
111             {
112                 if (s->move_up().key() == current.key())
113                     path += '^';
114                 else if (s->move_down().key() == current.key())
115                     path += 'v';
116                 else if (s->move_left().key() == current.key())
117                     path += '<';
118                 else
119                     path += '>';
120                 current = *s;
121                 s = s->parent;
122             }
123             for (int i = 0, j = path.length() - 1; i < j; i++, j--)
124             {
125                 auto k = path[i];
126                 path[i] = path[j];
127                 path[j] = k;
128             }
129             cout << path << endl;
130         }
131 };
132 const State_16 State_16::BAD(-1, -1, -1);
133 
134 ostream& operator << (ostream& out, const State_16& s)
135 {
136     if ((s.s & 0xfUL) == 0xfUL)
137         out << "   ";
138     else
139         out << setw(2) << (s.s & 0xfUL) + 1 << ' ';
140     for (unsigned long i = 4, m = 0xf0; m != 0; i += 4, m <<= 4)
141     {
142         if ((i & 0xf) == 0)
143             out << '\n';
144         if (((s.s & m) >> i) == 0xfUL)
145             out << "   ";
146         else
147             out << setw(2) << ((s.s & m) >> i) + 1 << ' ';
148     }
149     return out;
150 }
151 
152 namespace std
153 {
154     template<>
155     struct hash<State_16>
156     {
157         size_t operator()(const State_16& s) const
158         {
159             return s.key();
160         }
161     };
162 }
163 
164 template<class V>
165 class AStar
166 {
167 private:
168     unordered_set<V> closed;
169     class PriorityQueue : public priority_queue<V, vector<V>>
170     {
171     protected:
172         unordered_set<V> open_set;
173     public:
174         void push(const V& value)
175         {
176             auto index = open_set.find(value);
177             if (index == open_set.end())
178             {
179                 priority_queue<V, vector<V>>::push(value);
180                 open_set.insert(value);
181             }
182             else if (value.get_g() < index->get_g())
183             {
184                 *const_cast<V*>(&(*index)) = value;
185                 for (auto i = this->c.begin(); i != this->c.end(); i++)
186                     if (i->key() == value.key())
187                     {
188                         int pos = i - this->c.begin();
189                         __push_heap(this->c.begin(), pos, 0, value);
190                         break;
191                     }
192             }
193         }
194 
195         void pop()
196         {
197             open_set.erase(this->c.front());
198             priority_queue<V, vector<V>>::pop();
199         }
200     } open;
201 
```

```
202 public:
203     V perform_search(const V& initial)
204     {
205         closed.clear();
206         while (!open.empty()) open.pop();
207         open.push(initial);
208         V state;
209         const V *parent;
210         while (!open.empty())
211         {
212             state = open.top();
213             if (state.is_terminate())
214             {
215                 cout << "#visited : " << closed.size() << "\n#to visit: " << open.size() - 1 << endl;
216                 return state;
217             }
218             open.pop();
219             parent = &(*closed.insert(state).first);
220             for (V s : parent->children())
221             {
222                 s.set_parent(*parent);
223                 if (closed.find(s) == closed.end())
224                     open.push(s);
225             }
226         }
227         cout << "#visited : " << closed.size() << "\n#to visit: " << open.size() << endl;
228         return V::BAD;
229     }
230 };
231
232 int main()
233 {
234     State_16 s(0xfdceba9876543210UL, 15, 0), solution;
235     AStar<State_16> astar;
236     cout << -s.get_f() << endl << s << endl;
237     clock_t start = clock();
238     solution = astar.perform_search(s);
239     cout << (clock() - start) << endl;
240     if (solution.key() == solution.BAD.key())
241         cout << "No solution!" << endl;
242     else
243     {
244         cout << "f: " << -solution.get_f() << endl << solution << endl;
245         State_16::trace(solution);
246     }
247     return 0;
248 }
```