COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2018

[Assignment 2]
Solving Puzzle Games Optimally: Graph Search

Handed out: Wednesday, 3 of October
Due: 12:00 Noon, Wednesday, 17 of October

Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, your dexterity with efficient memory allocation and your algorithmic thinking, through programming a search algorithm over Graphs.
- Practice your ability to understand new algorithms based on those already taught in class.
- Gain experience with applications of graphs and graph algorithms to create an AI solver for a game.
- Foster your capability to understand a scientific paper and implement its ideas (optional).

Assignment description

In this programming assignment you'll be expected to build a *solver* for the 15–puzzle https://en.wikipedia.org/wiki/15_puzzle.

The 15–puzzle

These puzzles consist in assembling an image or some geometrical patterns, which has been decomposed into a number of sliding tiles. We can consider each tile to be identified by a number, so we can represent the initial state of the puzzle with the following diagram:

1	**2**	**7**	**6**
3	**4**	**5**	**8**
9	**10**	**11**	**12**
B	**14**	**15**	**13**

where B is a blank space. A puzzle solution is a sequence of moves which achieves the following state

B	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**
12	**13**	**14**	**15**

subject to the constraint that we can only swap positions of the blank tile with some adjacent one. For instance, in the initial state depicted above, the only valid moves would be:

1. Swap positions of tile 9 and blank B.
2. Swap positions of tile 14 and blank B.

The whole matrix is a state

Each possible configuration of the puzzle is called a state. The 15-puzzle Graph $G = \langle V, E \rangle$ is implicitly defined. The vertex set V is defined as all the possible puzzle configurations (states), and the edges E connecting two vertexes are defined by the legal movements, also called actions or operators.

Algorithm

Your solver should contain the implementation of the following search algorithm:

1. Iterative Deepening A^* (IDA*)

The algorithm follows the *Depth-First Search* search strategy and can be easily implemented as a recursive function.

IDA* has two parts:

1. The main loop, initializes the thresholds B and B' first. If no solution is found the search is triggered again but with an updated $B = B'$ threshold.
2. The recursive function that implements the bounded Depth-First Search.

In this assignment IDA* is guaranteed to find the optimal solution. In general, IDA* returns optimal solutions as long as the heuristic function is a lower bound on the true optimal solution length (see next section for more information on the heuristic used in this assignment).

Definitions and Comments about the Pseudocode

- The pseudocode assumes that you have a **Node** data structure – for variables n and n' – containing the fields s for the state, $g(n)$ for the cost of the path up to s from the initial state, and $f(n)$ for the evaluation function. Note that the notation $n'.g$ stands for the value of g function of node n'.
- **Node** - The data structure representing a possible path in the search. A node *may* have *children* nodes (corresponding to its possible successor states), a *parent* node (corresponding to the node from where the current node has been generated), and an operator index (recording which operator or action has been applied to generate the node, e.x. move blank left, right, up or down)
- **Successor State** - A successor state s' of state s is the resulting state of applying an action a.
- **Generated Nodes** - Nodes that have been created within the search. If you imagine the search algorithm building a tree, these are all the nodes of the tree.
- **Expanded Nodes** - Nodes for which its children have been generated. Here, it corresponds to the nodes that haven't been pruned by the threshold, or in terms of the search tree would be the internal nodes of the tree.
- **Pruning** - Evaluation of the path quality (node) and deciding whether we should not explore it further given a function $f(n)$ to evaluate and a threshold.
- $\mathbf{A}(\mathbf{s})$ is a function that returns the actions applicable in a given state s. In 15-puzzle it may be between 2 and 4 actions.
- The function $\mathbf{f}(\mathbf{a}, \mathbf{s})$ returns the state resulting of applying action a in state s.
- To print the **cost of the solution** found, you should look at $g(n)$ for node $n = r$ in the pseudocode.

- **h**(**s**) implements the heuristic function, taking as an argument the state s.
- Note that a node n is pruned only if $f(n)$ is strictly **bigger** than the threshold B.
- Note that in 15-puzzle the **cost function** of applying any action a in any state s is $c(a, s) = 1$.

```
IDA-Control-Loop()
 1  B ← h(s_0)
 2  repeat
 3          B' ← ∞
 4          n.s ← s_0
 5          n.g ← 0
 6          r ← IDA*(n, B, B')
 7          if r = NIL
 8            then B ← B'
 9    until r ≠ NIL
10  return r.g
```

Figure 1: Main search loop. s_0 stands for the initial configuration.

```
IDA*(n, B, B')
 1  for a ∈ A(n.s)
 2  do n'.s ← f(a, n.s)
 3     n'.g ← n.g + 1
 4     n'.f ← n'.g + h(n'.s)
 5     if n'.f > B
 6       then B' ← min(n'.f, B')
 7       else
 8            if h(n'.s) = 0
 9              then return n'
10            r ← IDA*(n', B, B')
11            if r ≠ NIL
12              then return r
13  return NIL
```

Figure 2: Recursive function implementing IDA*. If a solution is found, it returns the node r, otherwise returns NIL.

Heuristic

The heuristic that you will be using to prune the search space is the Sum of Manhattan Distances, which is defined as follows:

$$h(s) = \sum_i d(t_i, t_i^*) \tag{1}$$

where t_i is the position of the ith tile and t_i^* is the position where this tile should be. Positions are given by 2–component vectors $t_i = (x, y)$. Manhattan distance is easily computed

$$d(t_i, t_i^*) = |x - x^*| + |y - y^*| \tag{2}$$

For the initial state depicted above, the heuristic computation would be, starting from the upper left corner

$$h(s) = 1+1+2+2+4+1+1+4+1+1+1+4+1+1+2 = 27 \quad (3)$$

note that we don't take into account the (mis)placement of the blank space.

This heuristic is proved to be *admissible*, i.e. $h(s)$ is always lower or equal to the optimal distance $h^*(s)$ to the objective configuration from a any state s, for the 15–puzzle, being a useful heuristic to search for optimal solutions.

Deliverables, evaluation and delivery rules

Deliverable 1 – *Solver* source code

Just look for the **base code in LMS**. Understand the code and write your own code in the section where you see the comment: **FILL WITH YOUR CODE**.

You are expected to hand in the source code for your solver, written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command: `make` generating an executable called `15puzzle`. Remember to compile using the optimization flag `gcc -O3` for doing your experiments, it will run twice faster than compiling with the debugging flag `gcc -g`.

In order to test your solver, you can use the following 15–puzzle instances

expanded

N	Initial State s_0	$h(s_0)$	Cost	Nodes
1	14 13 15 7 11 12 9 5 6 0 2 1 4 8 10 3	41	57	276M
2	13 5 4 10 9 12 8 14 2 3 7 1 0 15 11 6	43	55	15M
3	14 7 8 2 13 11 10 4 9 12 5 0 3 6 1 15	41	59	565M
4	5 12 10 7 15 11 14 0 8 2 1 13 3 4 9 6	42	56	62M
14	7 6 8 1 11 5 14 10 3 4 9 13 15 2 0 12	41	59	1369M
88	15 2 12 11 14 13 9 5 1 3 8 7 0 10 6 4	43	65	6009M

where N is a unique ID from a test set of 100 initial states, s_0 is the initial state (first number corresponds with the tile placed in the upper left corner and **zero** represents the blank space), $h(s_0)$ is the value of the heuristic for the initial state, *Cost* is the optimal cost of the problem, and *Nodes* is the number of nodes for which their children are generated (it's a rough figure, your mileage might vary slightly depending on your implementation).

Your IDA* implementation will be required to find solutions of the same quality. With the data of the table you'll be able to asses the correctness of your implementation: it has to compute the exact same values for $h(s_0)$ and *cost*, with roughly a similar amount of *Nodes*.

Input

Your *solver* has to read the initial configuration from a file **with the same format as it appears in the table**, that is, a single line, containing a sorted list of indexes separated by a blank space. For example, instance $N = 1$ from the table would be:

```
14 13 15 7 11 12 9 5 6 0 2 1 4 8 10 3
```

output

Assuming we have a file called "1.puzzle" containting a single line with the configuration of instance $N = 1$ from the table, we will call our solver by running the following command:

`./15puzzle 1.puzzle`

and it will print into the `stdout` the following information:

1. Initial state of the puzzle.
2. $h(s_0)$ heuristic estimate for the initial state.
3. Thresholds the search have used to solve the problem.
4. Number of moves of the optimal solution.
5. Number of generated nodes.
6. Number of expanded nodes.
7. Number of expanded nodes per second.
8. Total Search Time, in seconds.

For example, the output of our solver `./15puzzle 1.puzzle` for instance $N = 1$ is:

```
Initial State:
14 13 15 7
11 12 9 5
6 0 2 1
4 8 10 3
Initial Estimate = 41
Threshold = 41 43 45 47 49 51 53 55 57
Solution = 57
Generated = 499,911,606
Expanded = 253,079,561
Time (seconds) = 5.12
Expanded/Second = 49,082,100
```

Deliverable 2 – Experimentation

Besides handing in the solver source code, you're required to provide a table, for your implementation of the algorithm and each of the problems in the table above, which accounts for

1. ID N of the Initial state of the puzzle.
2. $h(s_0)$ heuristic estimate for the initial state.
3. Thresholds the search have used to solve the problem.
4. Number of moves of the optimal solution.
5. Number of generated nodes.
6. Number of expanded nodes.
7. Number of expanded nodes per second.
8. Total Search Time, in seconds.

(optional) Explain concisely any improvements you made to speed up the algorithm in terms of number of expanded nodes per second, as well as any improvements you implemented suggested by the paper to reduce the number of expanded nodes.

Evaluation

Assignment marks will be divided into four different components.

1. IDA* (10)
2. Manhattan heuristic (2)
3. Experimentation (2)
4. Code Style (1)

Please note that you should be ready to answer any question we might have on the details of your assignment solution by e–mail, or even attending a brief interview with me, in order to clarify any doubts we might have.

Extra Mark

An extra mark can be earned if you implement any (1 at least) of the optimizations described in the the section Improved Admissible Heuristics of the article http://www.aaai.org/Papers/AAAI/1996/AAAI96-178.pdf written by the founder of IDA* Richard Korf.

If you do any of the optimizations for the extra marks, please report and discuss it in your experimentation.

Delivery rules

You will need to make *two* submissions for this assignment:

- Your C code files (including your `Makefile`) will be submitted through the LMS page for this subject: *Assignments* → *Assignment 2* → *Assignment 2: Code*.
- Your experiments report file will be submitted through the LMS page for this subject: *Assignments* → *Assignment 2* → *Assignment 2: Experimentation*. This file can be of any format, e.g. .pdf, text or other.

This double submissions process is being used because the program files need to be tested under UNIX, and the presence of non-text experiment files can interfere with the submit and verify process.

Program files submitted (Code)

Submit the program files for your assignment and your `Makefile`.

If you wish to submit any scripts or code used to generate input data, you may, although this is not required. Just be sure to submit all your files at the same time.

Your programs *must* compile and run correctly on the CIS machines. You may have developed your program in another environment, but it still *must* run on the department machines at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on the department machines at reasonably frequent intervals.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary. .

Advice and Hints

1. Create an initial puzzle that you know can be solved with few steps, e.g. take the goal state and just move the blank two positions. Your algorithm should be able to solve this problem quickly.
2. **If you allocate memory** for new nodes (Line 2 Fig 2), make sure to delete them as memory leaks might take all the space of your RAM.
3. Once you've got a working version, if you want to speed up your code **think carefully** where most of the computation goes. It's a good excuse to practice the art of profiling your code (understanding where most of the computation goes). Valgrind can be used for this purpose as well. Check documentation about it in this link. Most likely the most expensive operations will be computing the **heuristic** and **generating new nodes**. And just in case, we won't give extra marks or remove marks depending on your speed, but feel free to create a **Piazza** post with a ladder of the fastest programs. Of course be honest and feel free to share your programming tricks after the due date of the assignment.
4. Have in mind that some these problems might need – depending on hardware and implementation – more than 30 minutes to be solved. In order to test your code, just come up with a couple simple instances (simple as in "solutions requiring few moves"). You should be able to check by hand the solutions for those problems and assess the soundness of your implementation.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

"Borrowing" of someone else's code without acknowledgement is plagiarism, e.x. taking code from a book without acknowledgement. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic Honesty and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) on the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Administrative issues

When is late? What do I do if I am late? The due date and time are printed on the front of this document. The lateness policy is on the handout provided at the first lecture and also available on the subject LMS page. If you decide to make a late submission, you should send an email directly to the lecturer as soon as possible and he will provide instructions for making a late submission.

What are the marks and the marking criteria Recall that this project is worth 15% of your final score. There is also a hurdle requirement: you must earn at least 15 marks out of a subtotal of 30 for the projects to pass this subject.

Finally Despite all these stern words, **we are here to help!** There is information about getting help in this subject on the LMS pages. Frequently asked questions about the project will be answered in the LMS discussion group.