Assignment 1

Submit Deadline: 5pm 21st April 2023

1. Problem: Mini-SHRDLU

SHRDLU is a name of software written in late 1960s by Terry Winograd at MIT Artificial Intelligence (AI) Laboratory. The program was used to demonstrate how AI technologies could be used in natural language processing[1]. Inspired by, but different from, SHRDLU, **Mini-SHRDLU** is a computer game designed by Demis Hassabis, the CEO of Google DeepMind, and his research team for testing their AI algorithms[2].

A Mini-SHRDLU game contains k numbered blocks (from 1 to k respectively) on an $n*n$ vertically suspended board ($k \leq n^2-n$). All blocks have to be placed in the board, distributed in the columns, next each other down to the bottom in each column. A block on the top of a column can be moved from one column and deposited on the top of another column if space is available. At the beginning of each game, the board is randomly configured, called the *initial state.* A player of the game is given a *goal* or a set of *goals* to achieve by moving the blocks in sequence from the initial state. A goal can be any properties of the positions of blocks. For instance, we can require "*Block 1 to be at the bottom of the first column*" or "*Block 3 on the left of Block 2*". Any board configuration that satisfies the required goals is called a *goal state*. The following two figures show an example of initial state and goal state of a Mini-SHRDLU game with $k=6$ and $n=3$.

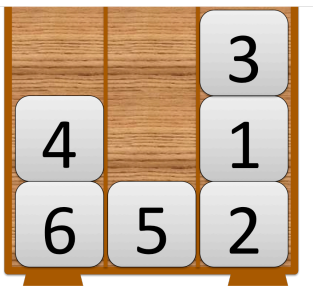

Figure 1: *An example of initial state*

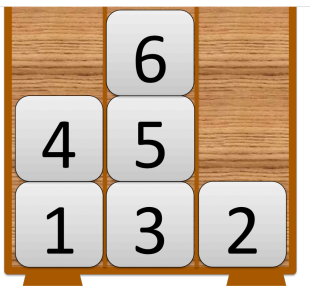

Figure 2: *An example of goal state*

This assignment is to develop a computer program that can host and automatically generate a solution to the Mini-SHRDLU game with any board size, n, and number of blocks, k, as long as $n>2$ and $n \leq k \leq n^2 - n$. However, for the simplicity of description, from now on we assume $k=6$ and $n=3$. The general case is quite similar.

---

[1] https://en.wikipedia.org/wiki/SHRDLU

[2] Alex Graves *et al*. Hybrid computing using a neural network with dynamic external memory, *Nature*, 471-476, 27 October 2016.

2. Game description

For a better understanding, we use the standard AI terminology to explain the game.

2.1 States

Any block configuration of a board is called a *state*. **Figures** 1&2 in the previous page are examples of states. A simple (but not necessarily the best) representation of a state is a two-dimensional array of integers in which zero stands for blank and the numbers 1 to 6 represent the blocks[3]. For instance, the following matrix represents the initial state shown in Figure 1.

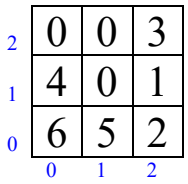

Figure 3: *The game state represents the game board of Figure 1. The blue figures are indices, not part of the state.*

Note that we use 0 to index the bottom row and the first column just because it is easier for implementation in an array or vectors.

2.2 Actions

An *action* is a move of the top block on one column to the top of another column. We use a pair of integers, (*source*, *destination*), to represent an action. For instance, (*2, 0*), means to move the top block in Column *2* to the top of Column *0*. An action is *legal* in a state if it is doable in that state. Given the state showing in **Figure** 3, the action (*2,0*) is legal, which moves Block 3 from column 2 to column 0 while (*0,2*) is illegal because column 2 is full. Neither is (1,2). In fact, only four actions are legal in that state: (*0,1*), (*1, 0*), (*2, 0*) and (*2,1*).

Once a game player applies a legal action to a state, the state will change to its *next state*. For instance, if we apply the action (*2,0*) to the state shown in **Figure** 3, its next state will be:

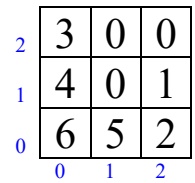

Figure 4: *The game state after applying action (2,0) to the state showing in **Figure** 3. Note that actions (1,0) and (2,0) are then not legal in this state.*

2.3 Goals

A *goal* can be any property of block configuration you want a game terminates with. For instance, you can require Block 1 to be located at (*0,0*). In this case, the goal can be represented as (*1, 0, 0*), meaning *Block 1 to be located at coordinate (0,0)*. A *goal*

---

[3] Other representation could be even better for the implementation of the game, for instance, an array of vectors or stacks, with pros and cons for each method. Feel free to use any data structure to represent states to best suit your solution.

state is any state that satisfies the goal. Normally one goal can have several goal states (why?).

A goal in the form (*block, row, column*) is called an *atom* goal, which means that the *block* must be located at location (*row, column*). Atom goals can be further combined in two different ways: disjunctive and conjunctive. If a set of atom goals $\{g_1, g_2, \ldots, g_m\}$ are *disjunctive*, any state that satisfies any of the atom goals is a goal state. For instance, the disjunctive goals {(1, 0, 0), (1, 1, 0), (1, 2, 0)} means Block 1 must be in Column 0 no matter which row it is. If a set of atom goals $\{g_1, g_2, \ldots, g_m\}$ are *conjunctive*, a goal state must satisfy all the atom goals in the set. For instance, the conjunctive goals {(1, 0, 0), (2, 0, 1), (3, 0, 2)} means Blocks 1,2&3 must be all in the bottom row in the goal state.

There are also other ways to represent goals. Let *'b', 'a', 'l'* and *'r'* denote '*below*', '*above*', '*left of*' and '*right of*' respectively. Then '*6b2*' means *Block 6 must be below Block 2*. '*4l1*' means *Block 4 must be at the left of Block 1*. These goals are named *neighbourhood goals*. Obviously neighbourhood goals can be transferred into combinations of disjunctive and conjunctive goals. Related tasks for neighbourhood goals are only for CS or BICT Advanced students or students who are thirsty for AI.

2.4 Plans

A *plan* is a sequence of legal actions that can bring a game from the initial state to a goal state. The task of this assignment is to design an algorithm that can automatically generate a plan to execute from any given initial state to a goal state if the goals are achievable. A typical way to find a plan is **game tree search** in term of Artificial Intelligence as shown in **Figure** 5. See the following for the pseudocode of game tree search algorithm.

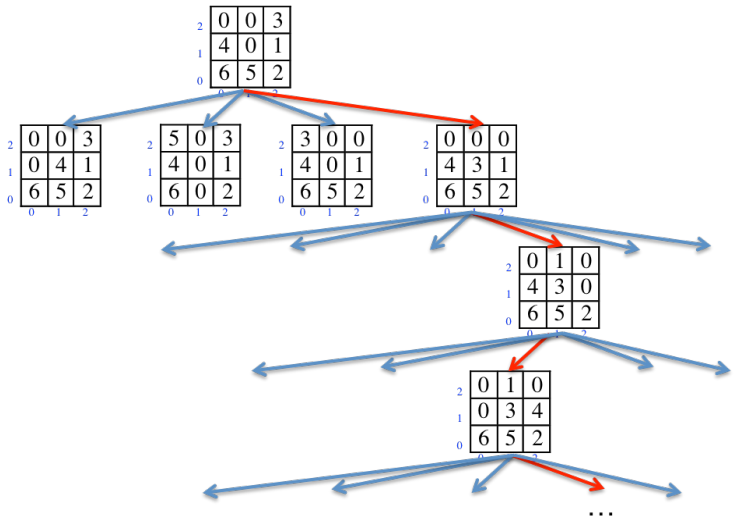

Figure 5: *An example of game tree search (only part of the search tree is displayed)*

```
Game_Tree_Search_Algorithm {
    fringe = {initial state};
    for (each state s in fringe) {
        if (s is a goal state) {
           return goal found;
```

```
        } else {
          for (each legal action in state s)
              Add all the next states of the action to
              fringe;
        }
    }
    return goal not found;
}
```

3. Tasks, requirements and hints

Through the development of a solution to Mini-SHRDLU, you will have a chance to practise with some of the data structures and algorithms learnt in the first six weeks, such as *vector*, *list*, *stack*, *queue*, *priority queue* and variety of algorithms. You are highly recommended to use C++ to implement your solution. Other general-purpose languages, such Java or Python, can be used. Ask your tutor before you start if you are going to use any language other than C++.

3.1 Pass Level

To gain a pass grade, you must complete the following tasks:

Task 1: *Create a class, named* State*, with related data members to store a game state (an array of nine integers, a 3*3 two-dimensional array of integers, an array of three vectors, an array of three stacks, or any other data types you prefer to use) and necessary methods for variable initialisation, data input and game board display.*

Task 2: *Implement methods in the* State *class or in any other appropriate classes to randomly generate an initial state.*

Task 3: *Implement methods in the* State *class or in any other appropriate classes to execute a legal action (remove a block from one column and deposit it to another column), output the list of all legal actions, and calculate the next state for any given action*.

Task 4: *Create a class, named* Solver[4]*, with at least a data member in type of* State *to record the current state of a game. Implement a search algorithm to find a goal state from any randomly generated initial state for any user input atom goal. Your search algorithm does not have to achieve any goal any time within a pre-set step limit (say 100 steps) but should avoid back and forth*.

Task 5: *Create a class driver containing the* main() *function to test all the classes you created.*

Requirements:

a. You may need to create other classes but your submission must contain at least the classes specified in the above tasks.

[4] The name does not matter. You may choose another name for the class.

b. Display of board states must be text based. No graphic output is needed. Keep the interface as simple as possible to avoid unnecessary complexity.
c. You must use either *stack*, *queue* or *priority queue* to store either legal actions, states or both. You may use the Standard Template Library or other ADT supplied by the IDE you use.
d. Your search algorithm must stop within 100 steps no matter whether the goal is achieved or not.

Hints:

a. Some tasks will be given as practical exercises, which will be good training for using generic data structures to store objects.
b. You might need to create a number of auxiliary methods in your state class to calculate the location of each block, the top block in each column and the deep of a block (how many blocks above the block) and so on.
c. For pass and credit level, your goal can be a single **atom goal** thus check if a state is a goal state can be very easy. Creating a class or a struct to represent an atom goal will make your life even easier.
d. You may need to overload == **operator** in the state class for sequential search in the list of all the visited states to avoid repeating.

3.2 Credit Level

To gain a credit grade, you must complete all the tasks for pass level plus the following task:

Task 6: *Create a heuristic function to evaluate the next state of each legal action so that the action that has more hope to achieve the goals has higher priority to be executed. Any other implementation, instead of a heuristic function, is also acceptable as long as it can make your search algorithm out of blind search.*

Requirements:

a. Use a priority queue to store the list of all legal actions at each state.
b. You don't have to guarantee success but your heuristic function should increate the chance of success.

Hints:

a. A heuristic function is normally associated with a state and a gaol, returning either an integer (typically between 0-100) or a real number (typically between 0-1). For instance, if a state has achieved the goal, you may return the full value 100. For another example, given the following state, if your goal is (1,0,0), you can give the following state a "half-way" value, say 50, because you only have to do two more move to empty column 0 in order to put 1 into its goal location:

2	0	1	0
1	4	3	0
0	6	5	2
	0	1	2

b. You might need to overload operator< for your Action class to use priority queue. Implement the comparator interface instead if you use Java.

3.3 Distinction Level

To gain a distinction grade, you must complete all the tasks for credit level plus the following tasks:

Task 7: *Accept user's input of a list of goals and try to solve them as disjunctive goals. Try to solve the first goal in 100 steps; if failed, try the next one, and so on.*

Task 8: *Accept user's input of a list of goals and check if these goals can be achieved in a single state (i.e., as conjunctive goals). Solve the goals as conjunctive goals.*

Requirements:

a. For disjunctive goals, you must display which goal has been achieved.
b. For conjunctive goals, all the atom goals in the given conjunctive goals must be achieved at the same goal state unless they are not achievable.

Hints:

a. Store all atom goals in a linear structure and call your search algorithm to try the goals in sequence.
b. It could be tricky to judge if an input of a set of goals are consistent. You may reject an input of new goal to avoid one block for different location or one location with different blocks.
c. For conjunctive goals, once an atom goal is satisfied, you have to maintain it while you try the other atom goals in the list. To achieve it, the ordering of goals to achieve is important.
d. You do not have to consider the case of disjunction of conjunctive goals or conjunction of disjunctive goals.

3.4 High distinction level

To gain a high distinction grade, you must complete all the tasks for distinction level plus the following tasks:

Task 9: *Rewrite your program so that it can process more general Mini-SHRDLU games where the board size, n, can be higher than 3 and the number of blocks can be any number between n and n^2-n.*

Task 10: Analyse *computational complexity of your search algorithm that finds a goal state from a given initial state and an atom goal.*

Requirements:

a. As you are targeting for high distinction, try your best to make your code with a professional looking -- appropriate abstraction, encapsulation, right use of inheritance and polymorphism, and so on.
b. Your complexity analysis must be written and submitted as a document.

Hints:

a. You do not have to test your code with too big board. You may assume $n<10$.

b. The documentation of complexity analysis must contain the following three parts:
   1) **The code of your algorithm**: you may present your algorithm by simplifying your search function in your program. If it calls other functions, please also include those functions. Ideally you rewrite your code in pseudocode to make it more readable.
   2) **Analysis of your algorithm:** Assume the size of board to be *n* and the number of blocks to be exactly n^2-*n* (the worst case of k). Pick up each loop in your algorithm and analyse their complexity in functions of *n* (approximation).
   3) **The complexity of your search algorithm in Big-O:** A summary of the complexity of your algorithm in Big-O notation.

c. If you use the standard data structures of STL, the complexity of each operation in the respective data structures can be found in the lectures. For priority queue, you may assume that *push()* and *pop()* are both in *O(log n)*.

Note: You do not need to analyse the complexity of your whole program. You only need to analyse the complexity of the algorithm that is used to find a goal state with any given initial state and an atom goal. For instance, you do not have to analyse how complicated to generate an initial state.

3.5 **Tasks for CS or BICT Advanced Activities or for students who want more challenges**

CS or BICT Advance students who choose to do their Advanced Activities with DSA are required to complete all the tasks 1-10 plus one of the following tasks:

Task 11 (Option 1): *Use A* search algorithm to find the minimal steps to achieve a goal*[5].

Task 11 (Option 2): *Improve your program so that you can process goals represented by the neighbourhood operators 'below', 'above', 'left of ' and 'right of '.*

Task 11 (Option 3): *Use any machine learning software, say TensorFlow, to implement a neurol network as a Mini-SHRDLU solver.*

Students other than CS/BICT Advance may also choose to do any of the above additional tasks to receive 10% of bonus marks for each task (capped with 100% for the assignment) to secure a top score.

4. Submission

All source code and documentation should be submitted via vUWS before the deadline for documentation purpose. Your programs (.h, .cpp or .java) can be put in separate files (executable file is not required). The document of your complexity analysis can be submitted in either Word file or PDF file. All these files, including your declaration text file, should be zipped into one file **with your student id as the zipped file name**. Submission that does not follow the format will not be accepted. **Email submission is not acceptable (strict rule).**

---

[5] https://en.wikipedia.org/wiki/A*_search_algorithm.

All students are required to submit a document (or as a comment in the file where the main function is), containing the following

DECLARATION

```
/*
I hereby certify that no other part of this submission has been copied from any
other sources, including the Internet, books or other student's work, or
generated from generative AI tools, such as ChatGPT except the ones I have
listed below:

// List the part of code you acquired from other resources

I hold a copy of this assignment that I can produce if the original is lost or
damaged.
*/
```

You do not have to declare if the code is from practical tasks, example code in the lectures, or anything supplied by the IDE or in the Standard Template Library. Contact your tutor if you are not sure.

4. Demonstration

You are required to demonstrate your program during **your scheduled** practical session in Week 8. Your tutor will check your code and your understanding of the code. **You will receive no marks if you fail the demonstration, especially if you miss the demo time.** Note that it is students' responsibility to get the appropriate compilers or IDEs to run their programs. You are allowed to run your program from your laptop at the specified demo time. **The feedback to your work will be delivered orally during the demonstration.** No further feedback or comments are given afterward.

The demonstrated program should be exactly the same as the one you submitted to vUWS. If you fail this assignment at your first demonstration, you are allowed to improve your work within the semester-break week (**maximal grade is 50% in this case**).

Again, do not send us your work via email. Programming doesn't work in that way. Face-to-face is the most efficient way to check your work.

Detailed marking criteria will be specified in a separate document.

Note: For a background of this game, you may download a video clip and an article from the assignment folder. However, the assignment does not rely on a full understanding of the content of these additional materials.

Have fun!