Computer Science, an Overview

Final Exam (A)

January 3, 2019

Department: Class: Name: Student No.:

# Read First Please

* Please fill in your **Department**, **Class**, **Name** and **Student No.**!
* The exam is divided into 4 parts, for a total of 100 points.
* You should work alone and **MUST NOT** copy others’.
* **The format of 8-bit floating-point notation**, **V8 machine language** and **the algorithm for a control system using heuristics** can be found on the reference page. You may tear the last page (empty) out as scratch paper.

# 1st PART: Single Choice (3’ for each question, 30’ in total) Write your choice for each question on the underline

1. Someone designed a flip-flop as follows to store a bit. Does this circuit satisfy all requirements to a flip-flop? \_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
|  |  |

1. No, it cannot set output to 0 regardless of the previous output.
2. No, the output is always 1.
3. No, no input can keep the previous output.
4. Yes, it does.
5. What are the results of and using the 8-bit floating point notation introduced in class? \_\_\_\_\_\_\_\_\_

|  |  |  |  |
| --- | --- | --- | --- |
| 1. , | 1. , | 1. , | 1. , |

1. What is the floating-point sum of 01101011 and 11011110 (the bit patterns represent values in 8-bit floating-point notation)? \_\_\_\_\_\_\_\_

A. 11011000 B. 01111001 C. 01011000 D. 00000001

1. Which of the following component in Von Neumann Architecture contains Program counter and Instruction register? \_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| 1. Bus | 1. CPU |
| 1. Memory | 1. Output devices |

1. After the computer is turned on, the CPU fetches the first instruction from \_\_\_\_\_\_\_\_

A. Program Counter B. Volatile Memory C. Read-only Memory D. Disk

1. Suppose each time slice in a **time-sharing** system is 50 ms and each context switch requires 3 μs (1ms = 1000μs). If there are a lot of processes, each of which executes an I/O request 1 μs after its time slice begins (assume OS suspends a process performing I/O operation and allows another process to run while the first is waiting), what **fraction** of the machine’s time is spent actually running these processes? \_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| 1. 99.998% | 1. 75% |
| 1. 50% | 1. 25% |
|  |  |

1. The following algorithm merges two sequential files. Assume that one input file contains records with key values equal to B, D and F while the other contains records with key values equal to A, C and E. How many records will be copied **before** executing the last line of the algorithm? \_\_\_\_\_\_\_\_\_

|  |
| --- |
| *// A procedure for merging two sequential files*  ***procedure*** *MergeFiles (lnputFile1, InputFile2, OutputFile)*  ***if*** *(both input files are at EOF)* ***then*** *(Stop, with OutputFile empty)*  ***if*** *(lnputFile1 is not at EOF)* ***then*** *(Declare its first record to be its current record)*  ***if*** *(lnputFile2 is not at EOF)* ***then*** *(Declare its first record to be its current record)*  ***while*** *(neither input file at EOF)* ***do***  *(Copy the current record with the "smaller" key field value to OutputFile;*  ***if*** *(that current record is the last record in its corresponding input file)*  ***then*** *(Declare that input file to be at EOF)*  ***else*** *(Declare the next record in that input file to be the file's current record)*  *)*  *Starting with the current record in the input file that is not at EOF, copy the remaining records to OutputFile.* |

|  |  |
| --- | --- |
| 1. 3 | 1. 4 |
| 1. 5 | 1. 6 |

1. What will the given Turing Machine do when its initial configuration is as the following (Xs are inputs to the Turing Machine which represent 0 or 1)? \_\_\_\_\_\_\_\_

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | \* | X | X | X | X | X | X | X | X | \* | \* |

↑

Initial State = START

Initial configuration:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Current State | Current Cell content | Value to write | Direction to move | New state to enter |
| START | \* | \* | LEFT | COMPUTE1 |
| COMPUTE1 | 1 | 1 | LEFT | COMPUTE1 |
| COMPUTE1 | 0 | 0 | LEFT | COMPUTE1 |
| COMPUTE1 | \* | \* | RIGHT | COMPUTE2 |
| COMPUTE2 | 1 | 1 | NO MOVE | HALT |
| COMPUTE2 | 0 | 1 | NO MOVE | HALT |
| COMPUTE2 | \* | \* | RIGHT | COMPUTE2 |

1. Set the most significant bit (i.e. the highest bit) of the input value to 1
2. Set the least significant bit (i.e. the lowest bit) of the input value to 1
3. Increment the input value by 1
4. Increment the input value by 0x80000000
5. Which of the following is an NP problem, but not yet known to be a P problem? \_\_\_\_\_\_\_\_
6. The Hanoi Tower problem
7. The Travelling Salesman problem
8. The Halting Problem
9. Searching a particular entry in a linked list of integers
10. Will a V8 machine executing the following program, starting from 0x00, halt? \_\_\_\_\_\_\_\_\_

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |
| Contents | 21 | 01 | 52 | 12 | B2 | 08 | B0 | 02 | C0 | 00 |

1. It will always halt.
2. It will never halt.
3. It will terminate in some cases but not always. Whether it will terminate depends on the initial contents in the general registers.
4. It is impossible to predict if the machine will halt even the initial contents in all general registers are known.

# 2nd PART: Eight-Puzzle (19’ for this part)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 |  | 1 | 2 | 3 |
| Start state |  | 4 | 6 | Goal state | 4 | 5 | 6 |
|  | 7 | 5 | 8 |  | 7 | 8 |  |

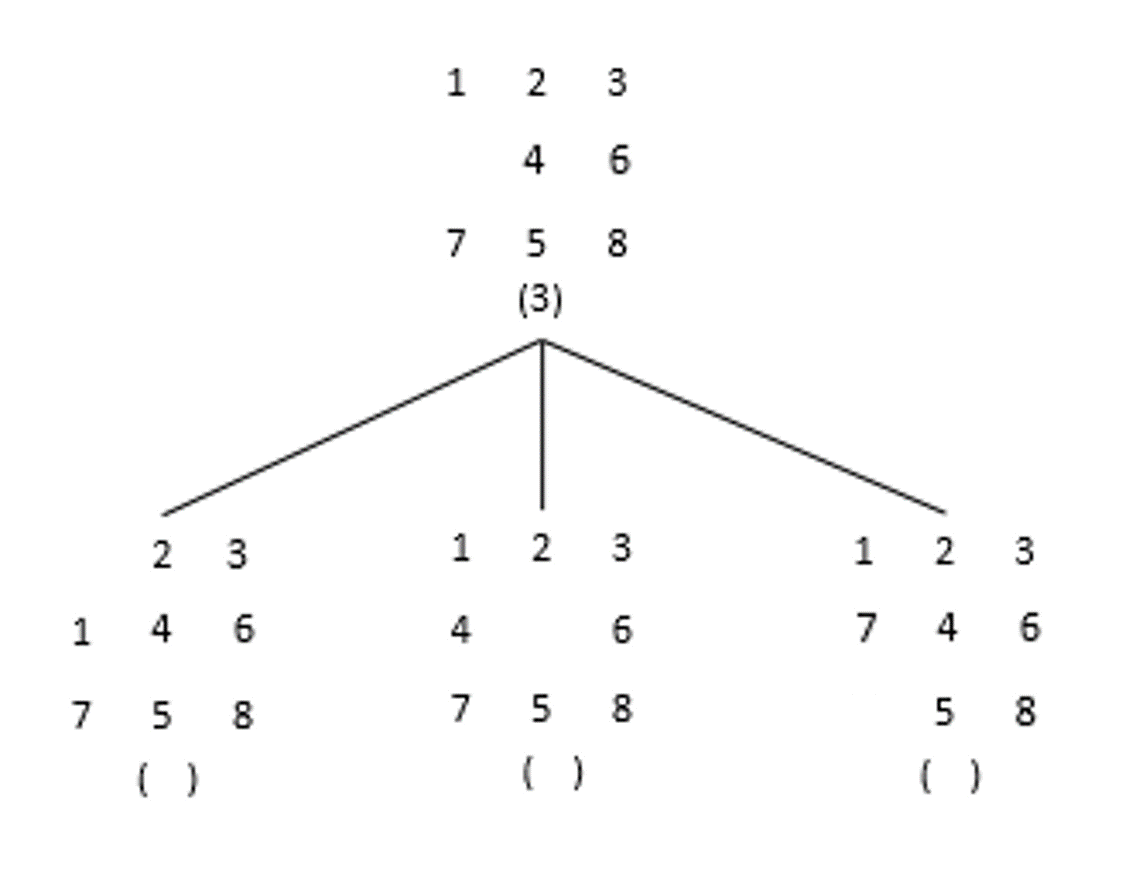
1. (3’) To implement an 8-puzzle solving algorithm, we need to think about how to store an eight-puzzle state. If each state is stored as a two-dimension array (the empty space is represented by 0), what is the index (started from 0) of the number in *i-th* row and *j-th* column (both started from 1) if the state is stored in row major order? If each element is an 8-bit integer, how many bytes are needed to save one state? Why?
2. (2’) The following algorithm builds a search tree in **a breadth-first manner**. The formal parameter *InitialNode* is the 8-puzzle we want to solve. What kind of data structure should be used in this algorithm?
3. Stack B. Queue C. Binary Tree D. Linked List

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **procedure** BreadthFirstSearch (*InitialNode*)  *ds* ← an empty \_\_\_\_\_\_\_  insert *InitialNode* into *ds*  **do** (  *s* ← The next element to remove from *ds*  remove *s* from *ds*  **if** (*s* is the goal node) **then**  (report a success and exit the current loop)  **else**  (insert all not-yet-visited nodes that can be reached by a single production into *ds*)  **end if**  ) **while** (**true**) |

1. (a) (2’) In 8-puzzle problem, how many states are accessible from the start state in any number of steps? How to determine whether two states are accessible to each other in any number of steps without searching?

(b) (2’) Answer the two questions in (a) if the board size change from 3\*3 to n\*m (digits: 0, 1, 2, …, n\*m-1).

1. (6’) Complete the **heuristic-based** search tree on the next page. The heuristic is **the number of out of place numbers**, i.e. the numbers that are not at the correct place where they should be in the goal state. For example, in the start state, ‘4’, ‘5’ and ‘8’ are out of place. Thus, the heuristic of the state is 3. Note that heuristic values of each state should also be **included** in the search tree. The algorithm can be found on the reference page.



1. (4’) Here is an algorithm for printing the solution to the 8-puzzle when the complete search tree is given. The search tree can be accessed via global variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **procedure** PrintSolutionAux (*n*)  **Print** the string “(enter Print)”  if (*n* is not the root of the search tree) then (  *p* ← The parent node of *n*  *op* ← production leading *p* to *n*  apply procedure PrintSolutionAux to p  **Print** the production *op*  )  **Print** the string “(exit Print)”  **procedure** PrintSolution()  *g* ← the **goal** node in the search tree  PrintSolutionAux(*g*) |

Productions are printed as ‘Move [the number] [direction]’, such as ‘Move 2 up’ or ‘Move 8 right’. What will be printed after executing the procedure PrintSolution on the search tree you have constructed in problem 4?

# 3rd PART: Stack (22’ for this part)

Stacks are widely adopted last-in, first-out (LIFO) data structures where *the last entry placed on a stack will always be the first entry removed*. This idea can be implemented in different ways. The following figure shows how to implement a stack using a linked list or a one-dimension array.

|  |  |
| --- | --- |
| A stack implemented using a linked list  (the stack is empty **if and only if** the head pointer contain the value NIL) |  |
| A stack implemented using a one-dimension array  (A stack introduced in class) |  |

Note that:

* In a stack implemented using a linked list, new elements are pushed **at the tail** of the list.
* In a stack implemented using an array, the stack pointer holds the index of the array entry where the top-most element in the stack can be found. The index starts from zero.

The structures of **LinkedList**, **Node** and **StackArray** in a linked list are given as follows.

|  |  |  |
| --- | --- | --- |
| struct StackList {  Node \*head;  } | struct Node {  int value;  Node \*next;  } | struct StackArray {  int arr[100];  int stack\_pointer;  } |

We will use the following notations in pseudo code.

* Node\* means the pointer to a Node.
* Int *arr*[100] means an array of 100 integers whose name is *arr*.
* The field **next** in the structure pointed to by the variable *node* is written as *node*.next.
* The *ith* element in an array *arr* is written as *arr[i]*.

The integers and pointers are 4-byte long each.

1. (4’) Each of the following statements compares the two different implementations of stacks from a certain point of view. Determine whether each of them is correct or not. Write a tick (✔) in the parenthesis if a statement is correct. Otherwise write a cross (✗).

Note: The *capacity* of a stack implemented using an array is the number of elements the reserved block of memory cells can hold.

( ) A stack implemented using one-dimensional array is not empty if and only if stack\_pointer equals to zero.

( ) The number of elements in a stack implemented using an array is limited to the size of the reserved memory block as long as we do not ask for a larger array after the stack is created. A stack implemented using a linked list, however, can grow as long as there is unused memory cells available.

( ) When there is only one element in the stack, stacks implemented using linked lists use more memory cells than stacks implemented using arrays with a capacity of 100.

( ) When there are 100 elements in the stack, stacks implemented using linked lists use more memory cells than stacks implemented using arrays with a capacity of 100.

1. (7’) Complete the following two procedures that push/pop an element to/from a stack implemented by a **linked list**. PushToStackByList insert the given Node (*node*) at the tail of the given StackList (stack). The pointer in *node* **should be properly initialized in the procedure**. PopFromStackByList removes the node at the tail of the given StackList (*stack*) if there is any node in it, or does nothing **if the stack is empty**. Please fill in the blanks with appropriate code to show how these procedures can be implemented by updating the pointers in the data structure. The procedures return nothing.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **procedure** PushToStackByList (*stack*, *node*)  **procedure** PopFromStackByList (*stack*) | |

1. (6’) Complete the following two algorithms that push/pop an element to/from a stack implemented by a one-dimension array. PushToStackByArray inserts the given Value (*value*) to the given StackArray (*stack*) if there is any space for growth, or does nothing **if the stack is full**. PopFromStackByArray removes a value from the given StackArray (*stack*) if there is any value in it, or does nothing **if the stack is empty**. Please fill in the appropriate code to show how these procedures can be implemented by updating the array and the *stack\_pointer*. PopFromStackByArray returns the removed value if the stack is not empty otherwise returns NIL. PushToStackByArray returns nothing.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **procedure** PushToStackByArray (*stack*, *value*)  **procedure** PopFromStackByArray (*stack*) | |

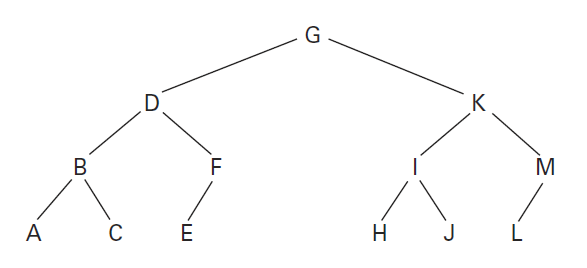
1. (4’) What is the time complexity of algorithms that push / pop an element into / from a stack? Complete the following table by filling the blanks in the rightmost column using the options given below. The in the options is the number of elements in the stack.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. The time cost is the same no matter what is | |  | |  |  |
| Type of stack | Operation | | Class in which the algorithm falls | | |
| Stacks implemented using linked lists | Push | |  | | |
| Pop | |  | | |
| Stacks implemented using arrays | Push | |  | | |
| Pop | |  | | |

1. (1’) How can we make a stack implemented using a linked list more efficient than the one we have discussed above?

# 4th PART: Binary Search Tree (29’ for this part)

A **binary search tree** is a binary tree with a value stored in each node. For any node in the search tree, the value stored in the node is greater than all values stored in its left sub-tree, and smaller than all values stored in its right sub-tree. The following figure shows how letters A through M can be stored in a binary search tree, assuming ‘A’ < ‘B’ < ‘C’ < … < ‘M’.



For example, consider the node storing the letter ‘D’. The left sub-tree of the node stores the letters ‘A’, ‘B’ and ‘C’ which are all smaller than ‘D’. The right sub-tree of the node stores ‘E’ and ‘F’ which are both greater than ‘D’.

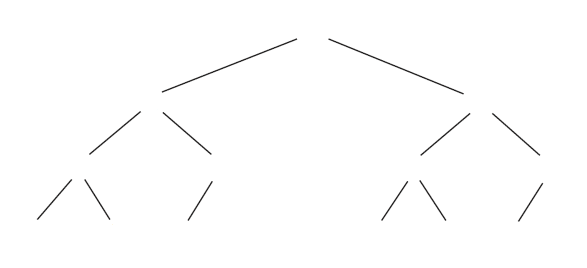
We can implement a binary search tree using a **one-dimension array**. In this part, we will store positive integers in the nodes and use -1 to indicate that the child does not exist. Given a **non-empty binary search tree** implemented in the above way, we can efficiently search for an entry using the following algorithm.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **procedure** Search (*Tree*, *TargetValue*)  *i* ← 0 (which is the index of the root node of *Tree,* as *Tree* is implemented as an array)  *v* ← 0  **while (true)** (  *v* ← The value stored at index *i* in *Tree*  **if** (*v* equals to *TargetValue* or *v* equals to -1) **then** ( break the loop )  **if** (*TargetValue* < *v*) **then**  (*i* ← the index of the left child of *i*, which is \_\_\_\_\_\_\_\_)  **else**  (*i* ← the index of the right child of *i*, which is \_\_\_\_\_\_\_\_)  )  **if** (*v* does not equal to -1) **then** (report the search to be a success)  **else** (report the search to be a failure) |

We now show you how the above algorithm can be implemented in V8. The following table shows the contents in the memory (all in hexadecimal notation). The content in a table cell is the bit pattern stored in the memory cell whose address is the sum of row and column headers of that table cell. For example, the lower right corner of the table says that the bit pattern FF is stored in the memory cell whose address is 0x70 + 0xF = 0x7F. Memory cells whose addresses range from 0x20 to 0x59 store the program, which **starts at 0x20**. Memory cells whose addresses range from **0x60 to 0x7F** store a binary search tree implemented as a one-dimensional array.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 20 | 21 | 01 | 22 | 02 | 23 | 80 | 24 | FF | 25 | 60 | 26 | 42 | 27 | 00 | 58 | 57 |
| 30 | 38 | 33 | 18 | 20 | 40 | 40 | B8 | 54 | 40 | 60 | B8 | 56 | 9A | 84 | 5A | A1 |
| 40 | 5A | A6 | 8A | A3 | 20 | 00 | BA | 4E | A7 | 07 | 57 | 71 | B0 | 2E | A7 | 07 |
| 50 | 57 | 72 | B0 | 2E | 27 | FF | 37 | 5F | C0 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 60 | 32 | 25 | 3D | 0C | 2D | 37 | 46 | 05 | 1A | 2B | FF | 33 | 3A | 42 | FF | FF |
| 70 | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF | FF |

1. (4’) The algorithm in pseudo code does not show how the left or right child of a node should be calculated. Complete the algorithm by filling the underlines on line 8 and 10 with the following options.
2. (6’) The following figure shows the shape of the binary search tree stored in the memory cells whose addresses start from **0x60**. Complete the tree by filling the integers stored in each node. You should keep the integers **in hexadecimal notation**.



1. The V8 program shown above can be divided into three parts, listed as follows.

* 0x20 to 0x2B: Code initializes some constants that are never changed in the program. **Register 6 stores *TargetValue* in the algorithm in pseudo code.**
* 0x2C to 0x53: Code implements the while loop part (from line 4 to line 11) of the algorithm. 0x2E to 0x33 implements the instructions to load “the value stored at index *i* in Tree”.
* 0x54 to 0x59: Code stores the result to memory cell whose address is 0x5F and halts.

1. (4’) In the V8 program, the *TargetValue* is stored in register \_\_\_\_\_\_\_\_ ; the address of the root node of the *Tree* is never changed, which is stored in register \_\_\_\_\_\_\_\_; register \_\_\_\_\_\_\_\_ acts as the index *i*, register \_\_\_\_\_\_ has the value stored in the node with index *i*.
2. (4’) The instructions from 0x34 to 0x3B determine the conditions on line 6 in the algorithm. If “*v* equals to *TargetValue*”, the contents in the program counter will be changed to \_\_\_\_\_\_\_\_. If “*v* equals to -1”, the contents in the program counter will be changed to \_\_\_\_\_\_\_\_.
3. (4’) The instructions from 0x3C to 0x47 compare the value stored in the node (i.e. *v*) with *TargetValue*. If *v* is greater than *TargetValue*, after executing the instruction in 0x42, the contents in register A will be \_\_\_\_\_\_\_\_. If *v* is smaller than *TargetValue*, the contents in register A will be \_\_\_\_\_\_\_\_.
4. (5’) If a V8 machine starts with 0x20 in the program counter and the above contents in memory, how many times will the instruction in memory cells whose addresses are 0x32 and 0x33 be executed when the machine halts? What are the contents in the memory cell whose address is 0x33 each time that instruction is executed? Answer the above questions by filling the following table. You can add more lines if you need.

|  |  |
| --- | --- |
| The # time instruction at 0x32 is executed | Contents in memory cell at 0x33 |
| 1st  2nd |  |

1. (2’) What is the content in the memory cell whose address is 0x5F after the machine halts?

**Reference page:**

# 8-bit Floating Point Notation

|  |  |  |
| --- | --- | --- |
|  | An Excess Four Conversion Table | |
|  | Bit Pattern | Value Represented |
| 111 | 3 |
| 110 | 2 |
| 101 | 1 |
| 100 | 0 |
| 011 | -1 |
| 010 | -2 |
| 001 | -3 |
| 000 | -4 |

# V8 Machine Language

|  |  |  |
| --- | --- | --- |
| **Op-code** | **Operand** | **Description** |
| **1** | **RXY** | **LOAD** the register R with the bit pattern found in the memory cell whose address is XY. |
| **2** | **RXY** | **LOAD** the register R with the bit pattern XY. |
| **3** | **RXY** | **STORE** the bit pattern found in register R in the memory cell whose address is XY. |
| **4** | **0RS** | **MOVE** the bit pattern found in register R to register S. |
| **5** | **RST** | **ADD** the bit pattern in registers S and T as though they were two’s complement representations and leave the result in register R. |
| **6** | **RST** | **ADD** the bit pattern in registers S and T as though they represented values in floating-point notation and leave the floating-point result in register R. The rightmost bits in the result will be lost if the result is too long to be filled into the mantissa field. E.g. 0.112 + 0.001112 will be 0.11112. |
| **7** | **RST** | **OR** the bit patterns in registers S and T and place the result in register R. |
| **8** | **RST** | **AND** the bit patterns in registers S and T and place the result in register R. |
| **9** | **RST** | **XOR** the bit patterns in registers S and T and place the result in register R. |
| **A** | **R0X** | **ROTATE** the bit pattern in register R one bit to the right X times. |
| **B** | **RXY** | **JUMP** to the instruction located in memory address XY if the bit pattern in register R is equal to the bit pattern in register 0. |
| **C** | **000** | **HALT** execution. |

# An algorithm for a control system using heuristics

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Establish the start node of the state graph as the root of the search tree and record its heuristic value  **while** (the goal node has not been reached) **do** (  Select the leftmost leaf node with the smallest heuristic value of all leaf nodes  To this selected node attach as children those nodes that can be reached by a single production  Record the heuristic of each of these new nodes next to the node in the search tree  ) |

**Scratch page:**

You are free to tear this page out and write anything on it. Please remember to hand this page in at the end of the exam.