CS2710 Homework1: An Evaluation Framework for Search Algorithms

October 6, 2009

Abstract

The Missionaries and Cannibals problem is formally formulated. Then, an evaluation framework enabling evaluating different search algorithms for different problems is presented. Under such framework, Iterative Deepening Search (IDS) algorithm and A* Search algorithm are implemented, in solving Missionaries and Cannibals problem. Both algorithms could adopt Tree or Graph Search strategy. Furthermore, we show that both algorithms function correctly and properly by comparing the output information with the theoretical result. Finally, we evaluate the performance of algorithms in terms of maximum size of fringe set, the number of nodes expanded and generated. Graph Search and Tree Search strategies are also compared.

1 Formulate Problem

1.1 Problem Formulation for Missionaries and Cannibals

The Missionaries and Cannibals problem ($M\&C$) is formulated as following. Assuming that there are m missionaries, c cannibals and the boat capacity is b.

- States: ordered sequence of three numbers representing the number of missionaries, cannibals and boats on the bank of the river from which they started.
- Initial state: $(m, c, 1)$.
- Goal test: test whether the goal state $(0, 0, 0)$ is reached.
- Successor function: take m_b missionaries and c_b cannibals across from $Side_1$ to $Side_2$ by boat, (assuming that currently there are m_{side1} missionaries and c_{side1} cannibals on $Side_1$, and m_{side2} missionaries and c_{side2} cannibals on $Side_2$) if:

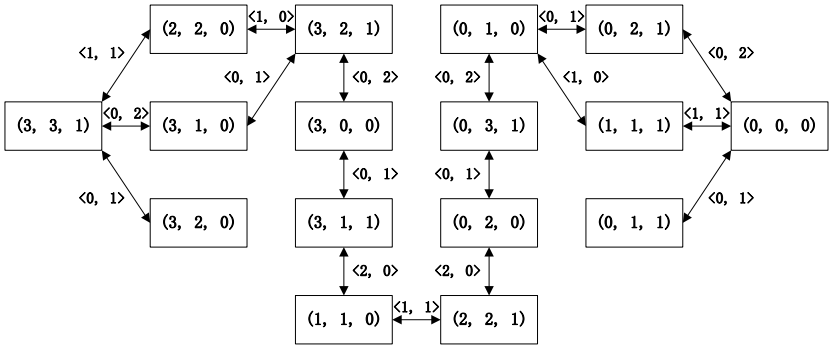

Figure 1: Complete state space for $M\&C$ where $m = 3$, $c = 3$ and $b = 2$

  - The boat is on $Side_1$, and at least 1, at most b people, including both missionaries and cannibals, could be on the boat, which means: $m_b + c_b \geq 1$ and $m_b + c_b \leq b$.
  - There are at least m_b missionaries and c_b cannibals on $Side_1$, which means: $m_b \leq m_{side1}$ and $c_b \leq c_{side1}$.
  - The missionaries on boat will not be overnumbered by cannibals, which means: $m_b = 0$ or $m_b \geq c_b$.
  - The remaining missionaries on $Side_1$ will not be overnumbered by cannibals, which means: $m_{side1} - m_b = 0$ or $m_{side1} - m_b \geq c_{side1} - c_b$.
  - The missionaries on $Side_2$ will not be overnumbered by cannibals, which means: $m_{side2} + m_b = 0$ or $m_{side2} + m_b \geq c_{side2} + c_b$.

- Cost function: number of crossings.

1.2 Complete State Space

Figure 1 shows the complete state space for $M\&C$ problem where $m = 3$, $c = 3$ and $b = 2$. Every vertex presents a possible state which presented in a form as specified in section 1.1. And each edge specifies the necessary action to transite from one state to another state, in a form of $< m_b, c_b >$, which means m_b missionaries and c_b cannibals will get on the boat to cross the river. For instance, there is a edge labeled with $< 1, 1 >$ between two verticies $(3, 3, 1)$ and $(2, 2, 0)$. We could interpret it as by taking 1 missionaries and 1 cannibals from the start side to the other side, the state trasites from $(3, 3, 1)$ to $(2, 2, 0)$, and by taking 1 missionaries and 1 cannibals from the other side to the start side, the state trasites from $(2, 2, 0)$ to $(3, 3, 1)$.

2 Node Generation

In this section, we will examine the sequence and order of nodes visited by Iterative Deepening Search (IDS) and A* algorithm. We assume that there are

3 missionaries and 3 cannibals, and the boat capacity is 2.

2.1 First 10 Nodes Visited by IDS

IDS could use Graph Search or Tree Search strategy, while Graph Search strategy eliminating reprocessing of nodes that have already been visited. The sequence and order of nodes visited by IDS are different when using different search strategy. We list the first 10 nodes (in order) visited by IDS for both Graph Search and Tree Search strategies.

- Using Graph Search strategy:

  (3, 3, 1), (3, 3, 1), (3, 2, 0), (3, 1, 0), (2, 2, 0), (3, 3, 1), (3, 2, 0), (3, 1, 0), (3, 2, 1), (2, 2, 0)

- Using Tree Search strategy:

  (3, 3, 1), (3, 3, 1), (3, 2, 0), (3, 1, 0), (2, 2, 0), (3, 3, 1), (3, 2, 0), (3, 3, 1), (3, 1, 0), (3, 2, 1)

2.2 Two Heuristics

Assuming that there are m_{side1} missionaries, c_{side1} cannibals and b_{side1} boats at the start bank, and the boat capacity is b. Note that b_{side1} could only be 0 or 1, since there is only one boat. The boat takes b people, but after each trip, one people must go back (to operate the boat). This will give us the following heuristic:

- Heuristic $H1$
  In order to get the heuristic, we try to solve a relaxed problem, by not taking into account the possibility of cannibals eating missionaires. The corresponding heuristic is expressed as below.

$$H_1() = \begin{cases} \max(1, 1 + 2 * \lceil \frac{m_{side1} + c_{side1} - b}{b-1} \rceil), & \text{if } m_{side1} + c_{side1} > 0 \text{ and } b_{side1} = 1 \\ \max(2, 2 * \lceil \frac{m_{side1} + c_{side1}}{b-1} \rceil), & \text{if } m_{side1} + c_{side1} > 0 \text{ and } b_{side1} = 0 \\ 0 & \text{if } m_{side1} + c_{side1} = 0 \end{cases}$$

- Heuristic $H2$
  We can further relax the problem, by assuming that the boat could go back to the start bank both automatically and without any cost after taking at most b people to the other bank. Such heuristic could be:

$$H_2() = \lceil \frac{m_{side1} + c_{side1}}{b} \rceil$$

2.2.1 Admissability of Heuristics

- Heuristic ($H1$)
  Heuristic $H1$ is admissible. If boat is at the start bank, one boat trip

(from start bank) would transfer b people to the other side at most. If boat is at the other bank, two boat trips (to start bank, and then from start bank) would transfer at most $b-1$ people to the other bank.

- Heuristic ($H2$)
  Heuristic $H2$ is also admissible, because one trip can take no more than b people from the start bank to the other bank.

2.2.2 Dominace Relationship Among $H1$ and $H2$

Heuristic H1 dominates H2, because $\forall$ state n, $H1(n) \geq H2(n)$.

2.3 First 10 Nodes Visited by A* with Herustic $H1$

As we explained in section 2.1, choosing different search strategy, Graph Search or Tree Search, would affect the sequence and order of nodes visited by search algorithms. We show the first 10 nodes (in order) visited by A* (with heuristic $H1$) for both Graph Search and Tree Search strategies.

- Using Graph Search strategy: (3, 3, 1), (3, 1, 0), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (3, 2, 0), (2, 2, 1), (0, 2, 0)
- Using Tree Search strategy: (3, 3, 1), (3, 1, 0), (2, 2, 0), (3, 2, 1), (3, 2, 1), (3, 0, 0), (3, 0, 0), (3, 1, 1), (3, 1, 1), (1, 1, 0)

3 Implementation

The program is implemented with Java programming language in Eclipse IDE.

3.1 Design

The whole project is organized to 5 packages.

- **Package edu.pitt.cs.cs2710.Problem** includes classes for defining problems which need to solve. This package currently have two classes, an abstract class Problem and another class called MCProblem. The Problem class defines the necessary interfaces (methods). The MCProblem class is a subclass of the Problem class, which formulate the $M\&C$ problem by defining behaviors for the corresponding interfaces (methods). We could later add new classes to define new problems, such as the 8-Puzzle problem. The important interfaces of these classes are listed. For simplicity, we do not repeatly explain the interfaces of a sub-class that inherit from its parent class. (This rule also applies for other packages.)
  - **Interfaces of the Problem class**
    - **abstract Node startNode();**
      It specifies the start node (initial state) of the problem.

    * **abstract boolean validate(Action action);**
      It checks whether an action is valid.
    * **abstract boolean validate(State state);**
      It checks whether a state is valid.
    * **abstract void print(int level);**
      It outputs the problem information.
    * **abstract List< Pair<Action, State> > successor(State state);**
      This method realizes the successor function, by returning all the possible successor states and the corresponding actions for a given state.
    * **abstract boolean goalTest(Node node);**
      This method implements the goal test function.
  - **Interfaces of the MCProblem class**
    * **Node startNode();**
    * **boolean validate(Action action);**
    * **boolean validate(State state);**
    * **void print(int level);**
    * **List< Pair<Action, State> > successor(State state);**
    * **boolean goalTest(Node node);**
    * **int boatCapacity();**
      Inquiry function for boat capacity.

- **Package edu.pitt.cs.cs2710.Search** includes classes for different search algorithms. This package currently have three classes, an abstract class Search and two sub-classes–AStarSearch and IterativeDeepeningSearch. The Search class defines the necessary interfaces (methods). AStarSearch and IterativeDeepeningSearch are two sub-classes of the Search class, which implement the A* and IDS algorithm. We could later add new classes to implement new search algorithms, such as the Depth-First and Width-First search. Actually, implementing more search algorithms is really simple, as will be explained later.
  - **Interfaces of the Search class**
    * **abstract Node search();**
      It is the interface for launching search to solve problems.
    * **Node execute(int depthLimit);**
      This is a template method which implements one time (pass)

search for specified depth limit (set to infinite if there is no depth limitation). Note that such method supports both Graph search and Tree search strategies. The search() method usually uses this method to implement the search algorithm. With the help of this existing method, implementing a new search algorithm would be very easy.

  - **void expand(Node node);**
    It expands a node according to the successor function of the problem.
  - **abstract Node getFromFringe();**
    It gets a node from the fringe set and defines the order in which the nodes in fringe set are processed and expanded. Different search algorithms would have different behavior for such method.
  - **abstract void addToFringeAll(List<Node> expandedNodes);**
    It adds a set of nodes to the fringe set and defines how to and in which order these nodes are added to the fringe set. Different search algorithms would have different behavior for such method.
  - **void initializeStatistics();**
    Initialize all the statistics.
  - **Heuristic heuristic();**
    Inquiry function for heuristic.
  - **boolean isGraphSearch();**
    Inquiry function for search strategy–Graph or Tree search.
  - **abstract void print(int level);**
    It outputs the statistics information about the search algorithm.

- **Interfaces of the AStarSearch and IterativeDeepeningSearch classes**
  - **Node search();**
  - **Node getFromFringe();**
  - **addToFringeAll(List<Node> expandedNodes);**

- **Package edu.pitt.cs.cs2710.Heuristic** includes classes for defining heuristics that are used by A* search algorithm. This package currently have three classes, an abstract class Heuristic and two sub-classes–MCHeuristicOne and MCHeuristicTwo. New heuristic rules could be incorporate into our program by simply adding a corresponding sub-class of Heuristic.
  - **Interfaces of the Heuristic class**
    - **abstract int h(State state, Problem problem);**
      It defines the heuristic function for the specific problem.
  - **Interfaces of the MCHeuristicOne and MCHeuristicTwo classes**

    - **int h(State state, Problem problem);**

- **Package edu.pitt.cs.cs2710.Util** includes several utility classes.
  - **The Node class** specifies the node information of the search tree or graph.
  - **The MCNode class** is a sub-class of the Node class that specifies the specialized node information for $M\&C$ problem.
  - **The State class** specifies the state information of a node.
  - **The MCState class** is a sub-class of the State class, and specifies the specialized state information for $M\&C$ problem.
  - **The Action class** specifies the action information.
  - **The MCAction class** is a sub-class of the Action class that specifies the specialized action information for $M\&C$ problem.
  - **The Print class** provides specialized facilities to print out information by setting up the importance level of each message to print.
  - **The Pair class** enables using pairs.
- **Package edu.pitt.cs.cs2710.Main** includes only one class, Main, to preprocessing the arguments, run the specific search algorithm on specific problem and print out the results.

3.2 Implementing Different Search Algorithms Using Function execute()

We will show the simplicity of implementing different search algorithms by using the execute() function, which is provided in the Search class of the edu.pitt.cs.cs2710.Search package. Listing1 lists the suedo code for the execute() function. One important parameter, depthLimit, specifies the depth limit of the one pass search.
Search algorithms could be implemented simply by launching the execute() function one time (for A*) or several times (for IDS). Listing2 and 3 show the suedo-code for implementing A* and IDS by using the execute() function. Note that, as specified in section 3.1, the search algorithm needs to define two other functions, getFromFringe() and addToFringeAll(), to specify how the fringe set should behave.

Listing 1: Seudo Code for execute() in class *Search*

```
Node execute(fringe, setAccessed, depthLimit){
    cutoffOccured = false;

    fringe.add(InitialNode);
    while (! fringe.isEmpty()) {
        node = GetFromFringe();
        if (GoalTest(node)) return node;
        if (node.depth() == depthLimit) {
            cutoffOccured = true;
            continue;
        }
        if (UseGraphSearch()) {
            if (! setAccessed.contains(node)) {
                setAccessed.add(node);
                AddToFringe(fringe, node.expand());
            } else {
                //replace the visited one by current node
                //if current node has a less path cost.
                nodeVisited = setAccessed.find(node);
                if (nodeVisited.g() > node.g()) {
                    setAccessed.remove(nodeVisited);
                    setAccessed.add(node);
                    AddToFringe(fringe, node.expand());
                }
            }
        } else {
            AddToFringe(fringe, node.expand());
        }
    }

    if (cutoffOccured) return CUTOFF;
    else return null;
}
```

Listing 2: Seudo Code for search() for class *AStarSearch*

```
Node search(){
    // initialize statistics
    initializeStatistics();
    // launching the search for once
    return execute(fringe, setAccessed, MAXIMUMINT);
}
```

Listing 3: Seudo Code for search() for class *IterativeDeepeningSearch*

```
Node search(){
    // initialize statistics
    initializeStatistics();

    // launching the search for several times with increasing depth limit
    for(int depthLimit = 0; depthLimit <= MAXIMUMINT; ++depthLimit) {
        Node resultDLS = execute(fringe, setAccessed, depthLimit);
        if (resultDLS != CUTOFF)
            return resultDLS;
    }

    return null;
}
```

3.3 Features

The important characteristics of the program are summariezed.

- Scalability: this program does not only implement IDS and A* search algorithms to solve $M\&C$ problem, but also create a framework which makes it very convenient and easy to add both new problems and new search algorithms.
- Both Graph Search and Tree Search strategies are supported. We could compare the effeciency difference between these two strategies.
- Comperhensive information are available for analysis. We could compare the performance and effeciency based on the information provided. And we could configure how much information to expose by setting up the verbose level. Four types of information are available.
  - Problem information: about the problem to solve.
  - Solution information: including the depth and path (actions) of the found solution.
  - Statistics about the search algorithm: including maximum length of fringe, # node generated, # node expanded, the average branching factor and # processed node with repeated state (only for Graph Search strategy).
  - Information about the search process: in terms of the sequence and order of nodes expanded. This information is only available in verbose mode.

3.4 Execution and Parameters

A jar file is submitted with the source code for the project. We can execute the program by launching the jar file using the following command.

$$java\ -jar\ Search.jar\ \text{“}<m\ c\ b>\text{”}\ algorithm\ heuristic\ useGraphSearch\ verboseLevel$$

The arguments are explained as follows.

- “$<mcb>$” describes the problem senario of $M\&C$ in terms of #missionaries, #cannibals, boat capacity.
- **algorithm** specifies the search algorithm to use–**ids** or **astar**.
- **heuristic** specifies the heuristic for A* algorithm–**none**, **h1** or **h2**.
- **useGraphSearch** is an optional argument which indicates whether to use Graph Search or Tree Search strategy–**0** stands for Tree Search strategy and **1** stands for Graph Search strategy. The default argument is 1 (Graph Search).
- **verboseLevel** is an optional argument which indicates the verbose level of the output information–**0**, **1** or **2**, where 0 stands for the succinct mode, and 2 stands for the verbose mode. The default argument is 1 (median mode).

4 Experiment Result and Evaluation

4.1 Sequence of Nodes Visited

In this section, we will check if the implemented search algorithms work correctly by comparing the sequence and order of nodes visited (expanded) against the theoretical results in section 2.1 and 2.3. The problem used in this section is the same as the one used in section 2.1 and 2.3, that there are 3 missionaries and 3 cannibals, and the boat capacity is 2.

4.1.1 IDS

- Using Graph Search strategy

  Listing4 shows the output information for IDS algorithm with Graph Search strategy. You can get such information by launch the program with command: ***java −jar Search.jar "<3 3 2>" ids none* 1 2**. The sequence and order of first 10 visited nodes perfectely match the result in section 2.1, and justisfy the correctness of the program.

- Using Tree Search strategy

  Listing5 shows only part of the output information for IDS algorithm with Tree Search strategy. Only the first 20 expanded nodes are listed, since the whole visit sequence is so huge (consisting of 8485 nodes). You can obtain the full output information by launch the program with command: ***java −jar Search.jar "<3 3 2>" ids none* 0 2**. The sequence and order of first 10 visited nodes perfectely match the result in section 2.1.

4.1.2 A* with Heuristic $H1$

- Using Graph Search strategy

  Listing6 shows the output information for A* algorithm with Graph Search strategy. You can get such information by launch the program with command: ***java −jar Search.jar "<3 3 2>" astar h*1 1 2**. The sequence and order of first 10 visited nodes perfectely match the result in section 2.3.

- Using Tree Search strategy
  Listing7 shows the output information for A* algorithm with Tree Search strategy. You can get such information by launch the program with command: ***java −jar Search.jar "<3 3 2>" ids none* 0 2**. The sequence and order of first 10 visited nodes perfectely match the result in section 2.1.

Listing 4: Program output for IDS algorithm with Graph Search strategy

```
===Problem Information===
  <3, 3, 2>

===Verbose mode information (sequence of EXPANDED nodes)===
  State: (3, 3, 1)  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (2, 2, 0)
  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (2, 2, 0)
  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (3, 0, 0)
  State: (2, 2, 0)  State: (2, 2, 0)  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)
  State: (3, 2, 1)  State: (3, 0, 0)  State: (3, 1, 1)  State: (2, 2, 0)  State: (2, 2, 0)
  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (3, 0, 0)
  State: (3, 1, 1)  State: (1, 1, 0)  State: (2, 2, 0)  State: (2, 2, 0)  State: (3, 3, 1)
  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (3, 0, 0)  State: (3, 1, 1)
  State: (1, 1, 0)  State: (2, 2, 1)  State: (2, 2, 0)  State: (2, 2, 0)  State: (3, 3, 1)
  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (3, 0, 0)  State: (3, 1, 1)
  State: (1, 1, 0)  State: (2, 2, 1)  State: (0, 2, 0)  State: (2, 2, 0)  State: (2, 2, 0)
  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (3, 0, 0)
  State: (3, 1, 1)  State: (1, 1, 0)  State: (2, 2, 1)  State: (0, 2, 0)  State: (0, 3, 1)
  State: (2, 2, 0)  State: (2, 2, 0)  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)
  State: (3, 2, 1)  State: (3, 0, 0)  State: (3, 1, 1)  State: (1, 1, 0)  State: (2, 2, 1)
  State: (0, 2, 0)  State: (0, 3, 1)  State: (0, 1, 0)  State: (2, 2, 0)  State: (2, 2, 0)
  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (3, 2, 1)  State: (3, 0, 0)
  State: (3, 1, 1)  State: (1, 1, 0)  State: (2, 2, 1)  State: (0, 2, 0)  State: (0, 3, 1)
  State: (0, 1, 0)  State: (0, 2, 1)

===Information about the found solution [Depth: 11]===
  0)Initial State          State: (3, 3, 1)
  1)Action: [0, 2, —>]       State: (3, 1, 0)
  2)Action: [0, 1, <—]       State: (3, 2, 1)
  3)Action: [0, 2, —>]       State: (3, 0, 0)
  4)Action: [0, 1, <—]       State: (3, 1, 1)
  5)Action: [2, 0, —>]       State: (1, 1, 0)
  6)Action: [1, 1, <—]       State: (2, 2, 1)
  7)Action: [2, 0, —>]       State: (0, 2, 0)
  8)Action: [0, 1, <—]       State: (0, 3, 1)
  9)Action: [0, 2, —>]       State: (0, 1, 0)
 10)Action: [0, 1, <—]       State: (0, 2, 1)
 11)Action: [0, 2, —>]       State: (0, 0, 0)

===Statistics for algorithm [IDS, h1, Graph Search]===
  1) Maximum length of fringe: 10
  2) # Node generated: 196
  3) # Node expanded: 92
  4) Average branching factor: 2.130434782608696
  5) # Node with repeated state: 77
```

Listing 5: Partial program output for IDS algorithm with Tree Search strategy

```
===Problem Information===
  <3, 3, 2>

===Verbose mode information (sequence of EXPANDED nodes)===
  State: (3, 3, 1)  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (2, 2, 0)
  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 3, 1)  State: (3, 1, 0)  State: (3, 2, 1)
  State: (3, 3, 1)  State: (2, 2, 0)  State: (3, 2, 1)  State: (3, 3, 1)  State: (3, 3, 1)
  State: (3, 2, 0)  State: (3, 3, 1)  State: (3, 2, 0)  State: (3, 1, 0)  State: (2, 2, 0)
  ... ...

===Information about the found solution [Depth: 11]===
  0)Initial State          State: (3, 3, 1)
  1)Action: [0, 2, —>]       State: (3, 1, 0)
  2)Action: [0, 1, <—]       State: (3, 2, 1)
  3)Action: [0, 2, —>]       State: (3, 0, 0)
  4)Action: [0, 1, <—]       State: (3, 1, 1)
  5)Action: [2, 0, —>]       State: (1, 1, 0)
  6)Action: [1, 1, <—]       State: (2, 2, 1)
  7)Action: [2, 0, —>]       State: (0, 2, 0)
  8)Action: [0, 1, <—]       State: (0, 3, 1)
  9)Action: [0, 2, —>]       State: (0, 1, 0)
 10)Action: [0, 1, <—]       State: (0, 2, 1)
 11)Action: [0, 2, —>]       State: (0, 0, 0)

===Statistics for algorithm [IDS, h1, Tree Search]===
  1) Maximum length of fringe: 17
  2) # Node generated: 19445
  3) # Node expanded: 8485
  4) Average branching factor: 2.2916912197996466
```

Listing 6: Program output for A* algorithm with Graph Search strategy

```
===Problem Information===
  <3, 3, 2>

===Verbose mode information (sequence of EXPANDED nodes)===
  State: (3, 3, 1)  State: (3, 1, 0)  State: (2, 2, 0)  State: (3, 2, 1)  State: (3, 0, 0)
  State: (3, 1, 1)  State: (1, 1, 0)  State: (3, 2, 0)  State: (2, 2, 1)  State: (0, 2, 0)
  State: (0, 3, 1)  State: (0, 1, 0)  State: (1, 1, 1)  State: (0, 2, 1)

===Information about the found solution [Depth: 11]===
  0)Initial State          State: (3, 3, 1)
  1)Action: [0, 2, —>]       State: (3, 1, 0)
  2)Action: [0, 1, <—]       State: (3, 2, 1)
  3)Action: [0, 2, —>]       State: (3, 0, 0)
  4)Action: [0, 1, <—]       State: (3, 1, 1)
  5)Action: [2, 0, —>]       State: (1, 1, 0)
  6)Action: [1, 1, <—]       State: (2, 2, 1)
  7)Action: [2, 0, —>]       State: (0, 2, 0)
  8)Action: [0, 1, <—]       State: (0, 3, 1)
  9)Action: [0, 2, —>]       State: (0, 1, 0)
 10)Action: [1, 0, <—]       State: (1, 1, 1)
 11)Action: [1, 1, —>]       State: (0, 0, 0)

===Statistics for algorithm [A*, h1, Graph Search]===
  1) Maximum length of fringe: 9
  2) # Node generated: 30
  3) # Node expanded: 14
  4) Average branching factor: 2.142857142857143
  5) # Node with repeated state: 9
```

Listing 7: Program output for A* algorithm with Tree Search strategy

```
===Problem Information===
  <3, 3, 2>

===Verbose mode information (sequence of EXPANDED nodes)===
  State: (3, 3, 1)  State: (3, 1, 0)  State: (2, 2, 0)  State: (3, 2, 1)  State: (3, 2, 1)
  State: (3, 0, 0)  State: (3, 0, 0)  State: (3, 1, 1)  State: (3, 1, 1)  State: (1, 1, 0)
  State: (1, 1, 0)  State: (3, 1, 1)  State: (3, 1, 1)  State: (1, 1, 0)  State: (1, 1, 0)
  State: (3, 2, 1)  State: (2, 2, 0)  State: (3, 2, 0)  State: (3, 3, 1)  State: (3, 3, 1)
  State: (2, 2, 0)  State: (2, 2, 0)  State: (3, 2, 1)  State: (3, 1, 0)  State: (3, 0, 0)
  State: (3, 2, 1)  State: (3, 0, 0)  State: (3, 1, 0)  State: (3, 0, 0)  State: (3, 1, 1)
  State: (3, 2, 1)  State: (3, 1, 1)  State: (1, 1, 0)  State: (3, 0, 0)  State: (1, 1, 0)
  State: (3, 1, 1)  State: (2, 2, 0)  State: (1, 1, 0)  State: (3, 2, 1)  State: (3, 3, 1)
  State: (3, 0, 0)  State: (3, 1, 0)  State: (3, 0, 0)  State: (3, 1, 0)  State: (2, 2, 0)
  State: (3, 1, 1)  State: (2, 2, 1)  State: (3, 2, 1)  State: (0, 2, 0)  State: (3, 2, 1)
  State: (3, 1, 1)  State: (3, 2, 1)  State: (1, 1, 0)  State: (3, 0, 0)  State: (3, 2, 1)
  State: (3, 1, 1)  State: (3, 0, 0)  State: (1, 1, 0)  State: (3, 1, 1)  State: (3, 2, 1)
  State: (1, 1, 0)  State: (3, 0, 0)  State: (3, 1, 0)  State: (3, 1, 1)  State: (3, 2, 1)
  State: (1, 1, 0)  State: (3, 0, 0)  State: (2, 2, 1)  State: (3, 1, 1)  State: (1, 1, 0)
  State: (0, 2, 0)  State: (1, 1, 0)  State: (0, 3, 1)  State: (1, 1, 0)  State: (0, 1, 0)
  State: (1, 1, 0)  State: (0, 2, 1)  State: (1, 1, 1)

===Information about the found solution [Depth: 11]===
  0)Initial State          State: (3, 3, 1)
  1)Action: [1, 1, —>]       State: (2, 2, 0)
  2)Action: [1, 0, <—]       State: (3, 2, 1)
  3)Action: [0, 2, —>]       State: (3, 0, 0)
  4)Action: [0, 1, <—]       State: (3, 1, 1)
  5)Action: [2, 0, —>]       State: (1, 1, 0)
  6)Action: [1, 1, <—]       State: (2, 2, 1)
  7)Action: [2, 0, —>]       State: (0, 2, 0)
  8)Action: [0, 1, <—]       State: (0, 3, 1)
  9)Action: [0, 2, —>]       State: (0, 1, 0)
 10)Action: [0, 1, <—]       State: (0, 2, 1)
 11)Action: [0, 2, —>]       State: (0, 0, 0)

===Statistics for algorithm [A*, h1, Tree Search]===
  1) Maximum length of fringe: 96
  2) # Node generated: 173
  3) # Node expanded: 78
  4) Average branching factor: 2.217948717948718
```

4.2 Evaluation of Algorithms, Strategies and Heuristics

4.2.1 Results

Table1 gives the maximum fringe size and the length of the found solution for different algorithms / strategies / heuristics. Note that a result labeled with -

Problem	Maximum fringe length								Length of found solution							
	Tree search strategy				Graph search strategy				Tree search strategy				Graph search strategy			
	IDS	A*(H1)	A*(H2)	A*(none)	IDS	A*(H1)	A*(H2)	A*(none)	IDS	A*(H1)	A*(H2)	A*(none)	IDS	A*(H1)	A*(H2)	A*(none)
<3 3 2>	17	96	3168	17694	10	9	5	5	11	11	11	11	11	11	11	11
<4 4 2>	-	-	-	-	7	11	6	5	-	-	-	-	*	*	*	*
<4 4 3>	23	208	2299	32234	15	22	16	12	9	9	9	9	9	9	9	9
<6 5 2>	-	94	-	-	19	41	14	14	-	19	-	-	19	19	19	19
<6 6 4>	31	568	8781	412578	26	42	27	25	9	9	9	9	9	9	9	9
<7 7 4>	38	996	105753	-	31	44	33	30	11	11	11	-	11	11	11	11

Table 1: The maximum length of fringe and the length of the found solution for different search algorithms / strategies / heuristics

Problem	# node expanded								# node generated							
	Tree search strategy				Graph search strategy				Tree search strategy				Graph search strategy			
	IDS	A*(H1)	A*(H2)	A*(none)	IDS	A*(H1)	A*(H2)	A*(none)	IDS	A*(H1)	A*(H2)	A*(none)	IDS	A*(H1)	A*(H2)	A*(none)
<3 3 2>	8485	78	2566	15096	92	14	14	14	19445	173	5733	32789	196	30	30	30
<4 4 2>	-	-	-	-	59	11	11	11	-	-	-	-	125	22	22	22
<4 4 3>	8164	106	1255	14191	124	15	19	20	24971	313	3553	46424	335	44	53	55
<6 5 2>	-	68	-	-	430	31	34	34	-	161	-	-	1009	75	79	79
<6 6 4>	51620	198	3566	114865	216	21	27	32	205772	765	12346	527442	758	77	99	111
<7 7 4>	939856	352	39450	-	363	29	36	38	3791002	1347	145202	-	1257	104	126	131

Table 2: The # node expanded and # node generated for different search algorithms / strategies / heuristics

means the specific algorithm could not finish the search process, may be due to out of memory problem (inefficiency). As you may know, the problem <4 4 2> does not have any solution. The result labeled with * means the algorithm detects that there is no solution.
Table2 gives the # node expanded and # node generated for different algorithms / strategies / heuristics.

The maximum fringe length could reflect the memory usage, and the # node expanded and generated could serve as indicators for the performance. We analyze the result as below.

4.2.2 IDS vs A*

- A* consumes much more memory than IDS, suggested by the maximum fringe length which implies the number of nodes need to keep in the memory for later expansion. A* algorithm with no heuristic, A*(none), is equivalent to Breadth First Search (BFS), and has the worst memory efficiency.

- A* is faster than IDS, implied by the # node expanded and generated. The performance difference between them is huge when the Tree Search strategy is applied.

4.2.3 Graph Search vs Tree Search

Graph Search strategy is much more effective and efficient (for both performance and memory) than Tree Search strategy.

- Graph Search strategy could effectively detect the situtation of no solution

exists, while the Tree Search strategy cannot find that because it will process repeated states and never stop.

- Graph Search strategy is much more efficient in terms of performance and memory usage than the Tree Search strategy. And Tree Search strategy sometimes cannot find the solution due to the performance and memory usage inefficiency.

4.2.4 H1 vs H2 vs none

- The goodness of heuristic impacts the performance and memory usage of the A* algorithm. When Graph Search strategy is applied, the performance and memory usage differences between different heuristics are insignificant, because there are very limited different states. Using Tree Search strategy, however, coule make the differences enormous. For instance, using Tree Search strategy, for problem <7 7 4>, A*(H1) is more than 100x memory efficient and almost 115x peroformance effective than A*(H2).
- H1 dominates H2, and therefore gets better performance and memory usage.
- Both H1 and H2 are admissible, and could always find an optimal solution.
- A*(none) is equivalent to DFS. Its memory usage scales exponentially with the problem size for the Tree Search strategy.

5 Conclusion

A scalable evaluation framework for search algorithms is implemented. The IDS and A* search algorithm, with Graph Search and Tree Search strategies, are implemented to solve the $M\&C$ problem under such framework. In addition, two heuristics for A* search algorithm are proposed.

We proved that the program works correctly by providing the output information and comparing it against the theoretical result in terms of the sequence and order of first 10 nodes visited.

Finally, we evaluate IDS and A* search algorithms with different search strategies and heuristics. We find that A* search can get better performance but worse memory usage, comparing to IDS. In addition, Graph Search strategy could make the search algorithm faster and less memory hungry. We also identifies that a good heuristic could make the A* search much more effecient. In our program, A* algorithm with Graph Search strategy and heuristic H1 is the best choice.