Homework 2
Missionaries and Cannibals

Instructor: Tsung-Che Chiang
tcchiang@ieee.org
Department of Computer Science and Information Engineering
National Taiwan Normal University

Problem Description

- Missionaries and Cannibals Problem
  http://www.learn4good.com/games/puzzle/boat.htm
Requirements

- Draw a diagram of the complete state space.
- Implement and solve the problem optimally using an appropriate search algorithm.
  - The optimality means the use of the smallest number of moves.
  - Show the solution step by step on the monitor.
- Report
  * what algorithm is used and
  * how many nodes are generated during search.

Submission

- Everyone should submit his/her own results.
- Title:
  - AI-HW2-your student ID(9 digits)
- Submit a .rar (or .zip) file including
  - your entire program (a single file is encouraged for the convenience of testing)
  - a document describing your search algorithm
  - the state-space diagram (NOT the search tree!)
- No re-submission is accepted.
Searching for Solutions

- General tree-search algorithm

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
fringe ← INSERT(Make-Node([INITIAL-STATE][problem]), fringe)
loop do
  if fringe is empty then return failure
  node ← REMOVE-FRONT(fringe)
  if GOAL-TEST([STATE][node]) then return SOLUTION(node)
  fringe ← INSERT-ALL([EXPAND(node, problem)], fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
for each action, result in SUCCESSOR-Fn([STATE][node]) do
  x ← a new NODE.
  PARENT-NO[x] ← node; ACTION[x] ← action; STATE[x] ← result
  PATH-COST[x] ← PATH-COST[node] + STEP-COST(node, action, x)
  DEPTH[x] ← DEPTH[node] + 1
  add x to SUCCESSORS
return SUCCESSORS
```

Implementation of Fringe

- BFS (queue, 1/2)

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main()
{
  queue<CNode> fringe;
  fringe.push(CNode("Arad"));
  cout << fringe.front() << endl;
  fringe.pop();
  fringe.push(CNode("Zerind"));
  fringe.push(CNode("Sibiu"));
  fringe.push(CNode("Timisoara"));
  cout << fringe.front() << ' ' << fringe.back() << endl;
  fringe.pop();
  return 0;
```

"Solving problems by searching," Artificial Intelligence, Spring, 2010
Implementation of Fringe

### BFS (queue, 2/2)

```cpp
class CState
{
public:
    CState(const string &loc):location_(loc) {}
    void print(ostream &os) const { os << location_; }
private:
    string location_; };

ostream & operator << (ostream &os, const CState &s) { s.print(os); }
class CNode
{
public:
    CNode(const string &loc):state_(loc) {}
    void print(ostream &os) const { state_.print(os); }
private:
    CState state_; };

ostream & operator << (ostream &os, const CNode &n) { n.print(os); }
```

### DFS (stack)

```cpp
int main()
{
    stack<CNode> fringe;
    fringe.push(CNode("Arad"));
    cout << fringe.top() << endl;
    fringe.pop();

    fringe.push(CNode("Zerind"));
    fringe.push(CNode("Sibiu"));
    fringe.push(CNode("Timisoara"));

    while (!fringe.empty())
    {
        cout << fringe.top() << endl;
        fringe.pop();
    }
    return 0;
}
```

*Solving problems by searching,* Artificial Intelligence, Spring, 2010
Implementation of Fringe

- **UCS (priority queue, heap, 1/2)**

```cpp
int main() {
    priority_queue<CNode, vector<CNode>, CNode::FNodeComp>
    fringe(CNode::hasGreaterPathCost);
    fringe.push(CNode("Arad", 0, 0));
    cout << fringe.top() << endl; fringe.pop();
    fringe.push(CNode("Zerind", 1, 75));
    fringe.push(CNode("Sibiu", 1, 140));
    fringe.push(CNode("Timisoara", 1, 118));
    cout << fringe.top() << endl; fringe.pop();
    fringe.push(CNode("Oradea", 2, 146));
    fringe.push(CNode("Arad", 2, 150));
    while (!fringe.empty()) {
        cout << fringe.top() << endl; fringe.pop();
    }
    return 0;
}
```

Implementation of Fringe

- **UCS (priority queue, heap, 2/2)**

```cpp
class CNode {
public:
    CNode(const string &loc, int depth, int pathCost):
        state_(loc), depth_(depth), pathCost_(pathCost) {}
    typedef bool (*FNodeComp)(const CNode &lhs, const CNode &rhs);
    static bool hasGreaterDepth(const CNode &lhs, const CNode &rhs)
        { return lhs.depth_ > rhs.depth_; }
    static bool hasGreaterPathCost(const CNode &lhs, const CNode &rhs)
        { return lhs.pathCost_ > rhs.pathCost_; }
    void print(ostream &os) const { state_.print(os); }
private:
    CState state_; int depth_, pathCost_;}
```

*Solving problems by searching,* Artificial Intelligence, Spring, 2010
Avoiding Repeated States

- If an algorithm remembers every state that it visited, then it can be viewed as exploring the state-space graph directly.

```cpp
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)

loop do
  if fringe is empty then return failure
  node ← REMOVE-FRONT(fringe)
  if GOAL-TEST[problem][STATE[node]] then return SOLUTION(node)
  if STATE[node] is not in fringe then
    add STATE[node] to fringe
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Implementation of Closed

```cpp
#include <set>
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

class CState
{
public:
  CState(const string &loc):location_(loc) {};

  bool operator < (const CState &rhs) const
  { return location_ < rhs.location_; }

private:
  string location_;}

int main()
{
  set<CState> closed;
  closed.insert(CState("Arad");

  if (closed.find(CState("Arad")) != closed.end())
    cout << "Arad is already closed." << endl;
  return 0;
}
```

"Solving problems by searching," Artificial Intelligence, Spring, 2010