Xây dựng Agent giải bài toán FreeCell

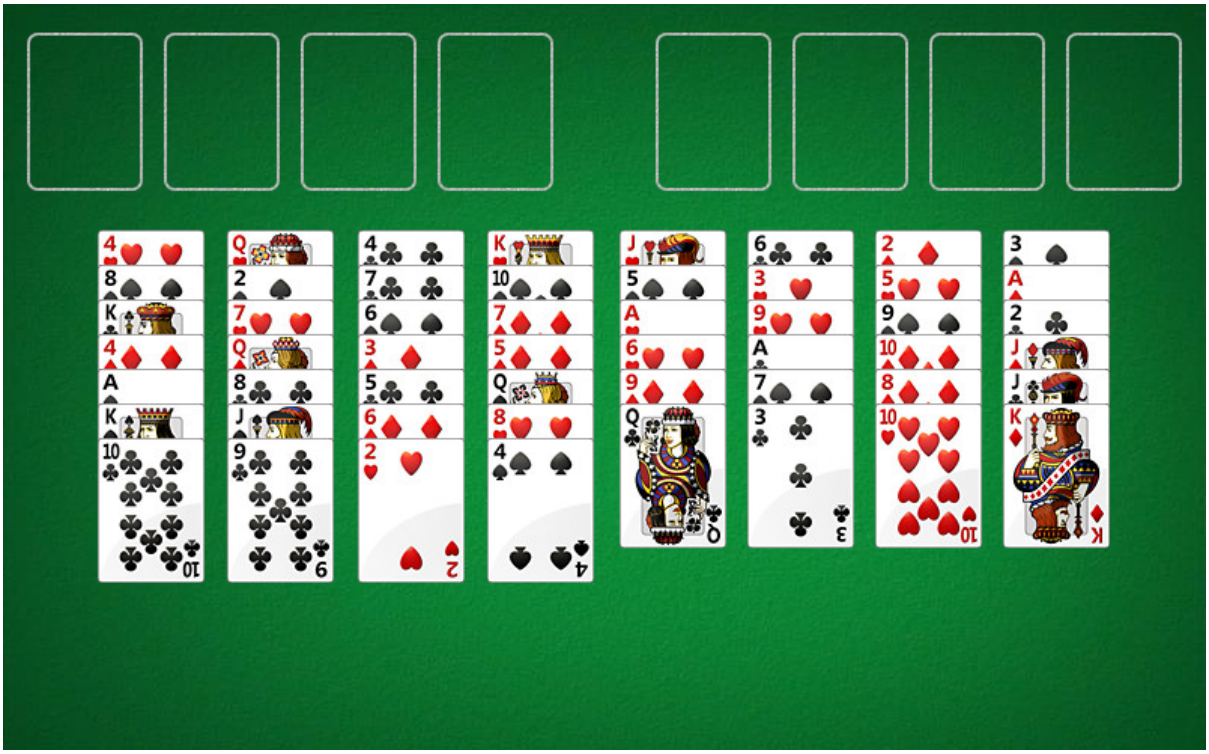
Giảng viên hướng dẫn: PGS.TS. Bùi Thế Duy

Thành viên: Nguyễn Hoàng Sơn, Nguyễn Hà Anh Tuấn, Nguyễn Đình Tư

Mục lục

1. Giới thiệu bài toán FreeCell

FreeCell là một trò chơi bao gồm 52 quân bài khá phổ biến trong hệ điều hành Windows của Microsoft. Các quân bài trong FreeCell đều được lật lên để người chơi xác định được trạng thái cũng như vị trí của các quân bài ngay từ đầu.



Hình 1. Trạng thái khởi đầu của FreeCell

Hình bên trên là giao diện của trò chơi FreeCell trên hệ điều hành Windows Vista. Tại trạng thái khởi dầu của trò chơi, có 4 ô *FreeCell* trống và 4 ô *Foundation* trống lần lượt là các ô nằm phía trên trong hình. 52 quân bài được chia thành 8 cột, 4 cột có 7 quân ở bên trái và 4 cột còn lại có 6 quân. Mục tiêu của FreeCell là đưa tất cả 13 quân cùng chất với nhau lần lượt từ A, 2, 3, …, 10, J, Q, K lên các ô *Foundation* tương ứng với chúng. Trò chơi kết thúc khi tất cả 4 quân K của các chất đều được đưa lên *Foundation*. Trong FreeCell để chuyển quân bài *b* đặt lên trên quân bài *a* chúng phải thỏa mãn điều kiện:

$$a \text{ nằm dưới } b \Leftrightarrow \begin{cases} rank(a) = rank(b) + 1 \\ isblack(a) \neq isblack(b) \end{cases}$$

trong đó:

$rank(a) = \{1, 2, ..., 12, 13\} \Leftrightarrow \{A, 2, 3, ..., 10, J, Q, K\}$, đây là giá trị của quân bài khi loại bỏ chất của nó

$isblack(a) = \begin{cases} true \Leftrightarrow a \in \{spade, club\} \\ false \Leftrightarrow a \in \{heart, diamond\} \end{cases}$ việc này tương đương với kiểm tra quân bài là quân bài đen hay quân bài đỏ

Chẳng hạn, như hình trên, chúng ta có thể chuyển quân *9 tép* ở cột *2* sang cột *7* nhưng không thể chuyển quân *9 tép* từ cột *2* sang cột *1* hoặc các cột còn lại.

Nhìn chung về luật chơi, FreeCell có 4 loại di chuyển cơ bản chính:

- Chuyển 1 quân bài trên cùng của cột *x* sang cột *y*.
- Chuyển 1 quân bài *a* lên *Foundation*. Việc chuyển quân bài lên *Foundation* chỉ thực hiện được khi quân bài đó là quân *A* hoặc đã có quân bài *b* có *rank(b)+1 = rank(a)*, và *a* với b đồng chất trên *Foundation*
- Chuyển 1 quân bài lên *FreeCell*, việc chuyển này chỉ thực hiện khi *FreeCell* vẫn còn chỗ trống. sau mỗi lần chuyển, số ô trống trên *FreeCell* giảm đi 1
- Chuyển 1 quân bài từ *FreeCell* xuống một cột nào đó, việc chuyển này tuân theo luật ở trên, đồng thời sau khi chuyển FreeCell chứa quân bài đó sẽ trống

Chúng ta có tất cả *52* quân bài điều này dẫn tới chúng ta sẽ tạo ra *52!* trạng thái khởi tạo, tuy nhiên do các cột có thể được đổi vị trí với nhau đồng thời chúng ta chỉ cần xét xem quân bài là quân đỏ hay đen khi di chuyển dẫn tới những trường hợp bị lặp. Sau khi loại bỏ các trường hợp trên chúng ta sẽ có khoảng $1.75*10^{64}$ trạng thái khởi tạo ban đầu cho FreeCell. Hiện tại với hệ điều hành Windows mới nhất của Microsoft chúng ta có *1000000* trạng thái bắt đầu trong đó có tồn tại những trạng thái mà chúng ta vẫn chưa thể tìm được thuật toán tốt để giải bài toán với chỉ 4 *FreeCell* chẳng hạn như trạng thái số *11982*

2. Cấu trúc agent

Để xây dựng được một agent giải bài toán FreeCell, chúng ta cần đi qua các bước: nhận dạng trạng thái ban đầu, xác định tất cả các bước đi có thể cho 1 trạng thái nào đó, tìm kiếm lời giải, thực hiện lời giải

2.1. Nhận dạng trạng thái ban đầu

Đầu tiên chúng ta có thể dễ dàng nhận thấy môi trường của agent giải bài toán FreeCell có các tính chất sau

- *Accessible*: Agent có thể dễ dàng xác định trạng thái của các quân bài trong trò chơi FreeCell do tất cả các quân bài đều được lật lên

- *Static*: Trạng thái của các quân bài chỉ bị thay đổi khi agent thực hiện một bước nào đó khi đã xác định được lời giải
- *Deterministic*: Mỗi một hành động của agent đều tạo ra một trạng thái mới, trạng thái mới này sẽ được agent dự đoán trước một cách rõ ràng

Ở đây, chúng tôi sẽ sử dụng một cách đơn giản nhất đó là để người dùng nhập trạng thái của các quân bài dưới dạng text sau đó để agent xác định môi trường mà nó được đưa vào. Hình dưới là một ví dụ về input của người dùng(cho game 31645)

```
.. .. .. .. .. .. .. ..
4H QH 6D QD 3S 5S 9C 5C
JH 8C AD AH JC 2H 5H AS
9H TS JS 8S KH 2C 9S 7C
JD 4S 7H 4D 2D 7D 4C KD
TC 7S 6H 5D 8D 3C 6S 8H
6C KC TD TH AC 2S 3H QS
KS 3D QC 9D
```

Hình 2. Input của game 31645

Trong đó, các kí tự đầu tiên là rank của quân bài, kí tự tiếp theo là chất của quân bài. H tương ứng với chất cơ, D tương ứng chất rô, C tương ứng chất bích, và S tương ứng chất tép. Mỗi điểm có dấu ".." tương ứng với vị trí đó bỏ trống. Hàng đầu tiên của input biểu diễn trạng thái của các ô FreeCell và các ô Foundation, các hàng còn lại là các quân bài tại 8 cột tương ứng. Người chơi có thể đưa 1 trạng thái bất kì vào để agent tìm lời giải, chẳng hạn người chơi có thể đưa vào một trạng thái đơn giản như sau

```
.. .. .. .. 6C 8D 9H 7S
QC JC 8S KS QD QH QS JS
KC 7C 9S TC JH 8C TH JD
KH KD TS .. TD 9D 9C
```

Hình 3. Một input khác

Nhằm tối ưu thuật toán tìm kiếm cho agent, các trạng thái đều được mã hóa nhằm mục đích so sánh (với các trạng thái đã xảy ra trước thì không cần tìm lời giải từ trạng thái đó nữa)

2.2. Xác định các bước đi có thể

Tại mỗi trạng thái sau khi thực hiện một hành động, agent cần tìm tất cả các nước đi

tiếp theo có thể có của nó. Ở đây tất cả các bước đi đều được liệt kê ra nhằm mục đích tìm kiếm con đường tốt nhất.

Ngoài ra, agent có thể tìm kiếm 1 số bước tắt hoặc một số bước thực hiện tự động nhằm làm giảm sự bùng nổ của trạng thái trong quá trình tìm kiếm:

- Một bước thực hiện tự động là bước chuyển 1 quân bài lên Foundation khi tất cả các quân bài có thể đặt lên trên nó đã được đưa lên Foundation
- Một bước tắt là một bước chuyển n quân bài từ cột này sang cột kia với $n \geq 1$. Giả sử tại 1 trạng thái bất kì chúng ta có x ô *FreeCell* trống và y cột trống. Khi đó:
  + Nếu chúng ta di chuyển từ 1 cột tới 1 cột khác không trống ta sẽ có $n \leq (x+1)(y+1)$
  + Nếu chúng ta di chuyển từ 1 cột tới 1 cột hiện tại không có quân bài nào ta sẽ có $n \leq (x+1)y$

2.3. Tìm kiếm lời giải

2.3.1. Thuật toán tìm kiếm "lai"

Chúng tôi sử dụng thuật toán tìm kiếm "lai". Đây là thuật toán tìm kiếm kết hợp giữa tìm kiếm theo chiều rộng và tìm kiếm theo chiều sâu. Thuật toán có lưu trữ và kiểm tra các trạng thái đã sinh ra bằng cách mã hóa các trạng thái đồng thời còn xét đến cả độ ưu tiên theo tiêu chí đánh giá từng trạng thái trò chơi.

Ưu điểm của thuật toán

- Kết hợp được ưu điểm của thuật toán tìm kiếm chiều rộng và chiều sâu. Tìm kiếm theo chiều rộng giúp chúng ta có thể rút ngắn được số bước đi để đạt được trạng thái kết thúc của trò chơi. Tìm kiếm chiều sâu giúp chúng ta tránh khỏi việc bùng nổ không gian trạng thái và có thể nhanh tìm ra được trạng thái kết thúc.
- Mã hóa các trạng thái để lưu trữ và kiểm tra giúp chúng ta tiết kiệm được bộ nhớ cần cho việc lưu trữ trạng thái và tránh được nhiều trạng thái lặp.
- Việc đánh giá, cho điểm cho từng trạng thái làm cho các trạng thái tiềm năng, các trạng thái gần đạt đến trạng thái kết thúc, được xét ưu tiên sớm, rút ngắn thời gian tìm kiếm.

Đầu vào

- Trạng thái ban đầu của trò chơi
- Hàm đánh giá trạng thái

Đầu ra

- Kết quả là có tìm được cách chơi để thắng hay không, nếu có trả ra danh sách các bước đi.

Mô tả thuật toán

- Thuật toán sử dụng *một hàng đợi với độ ưu tiên (pQueue)* theo hàm đánh giá $\alpha(s)$ để lưu các trạng thái sẽ được "thăm" tiếp theo. Theo tiêu chí của hàm ưu tiên, các trạng thái tiềm năng hơn sẽ ở đầu hàng đợi để được "thăm" trước.
- Thuật toán sử dụng *một cây từ điển (prev)* để lưu trữ và kiểm tra các trạng thái đã sinh ra.
- Thuật toán sử dụng *một ngăn xếp (moveStack)* để lưu các bước đi.
- Đầu tiên, trạng thái ban đầu được đẩy vào hàng đợi và thuật toán lần lượt tìm kiếm cho đến khi nào hàng đợi rỗng. Khác với tìm kiếm theo chiều rộng thông thường, ở mỗi bước, thuật toán "lai" này lấy một trạng thái ra khỏi hàng đợi và thực hiện tìm kiếm theo chiều sâu với độ sâu tối đa là K (trong trường hợp này K=6) bằng cách sinh ra các trạng thái mới từ trạng thái đó bằng các luật chơi. Các trạng thái sinh ra ở bước tìm kiếm theo chiều sâu K này được kiểm tra xem có phải trạng thái đích hay không, nếu đúng thì kết thúc tìm kiếm, nếu không thì xét xem trạng thái sinh ra đã có độ sâu là K hay chưa, nếu chưa thì tiếp tục đệ quy tìm kiếm với trạng thái này, nếu đã có độ sâu là K thì đệ quy dừng lại và đẩy trạng thái này vào trong hàng đợi ưu tiên *pQueue*. Các trạng thái sinh ra đều được xét và đẩy vào trong *prev* đồng thời các bước đi được lưu lại. Đây chính là bước thể hiện tính "lai" của thuật toán, kết hợp giữa tìm kiếm chiều rộng có độ ưu tiên và tìm kiếm chiều sâu có giới hạn. Chúng tôi giới hạn không gian lưu trữ của *prev* với khoảng tối đa hai triệu phần tử. Khi *prev* vượt quá, chương trình sẽ xóa toàn bộ prev và tiếp tục lưu lại từ đầu.
- Mã giả:

visit(currentState, depth)

```
if currentState is ending state
      return successful;
end if
if depth > K
      encode currentState to a key.
    put the key to prev.
      evaluate currentState by scorer
      add currentState to pQueue with priority of scorer
      return unsuccessful;
end if
for-each valid move
      tranform currentState to nextState by the move
      encode nextState to a key
      put the key to prev
      put the move to moveStack.
      visit(nextState, depth+1)
end for-each
```

solve(startedState, scorer)

```
Init prev, moveStack, pQueue;
pQueue ← startedState;
while pQueue is not empty
    currentState ← pQueue.pop();
    visit(currentState, 0);
    if visit(currentState, 0) is successful
          save last moves to currentState
          traceback(currentState);
          return true;
    end if
end while
```

2.3.2. Mã hóa các trạng thái

Chúng tôi coi các trạng thái chính là một dãy các lá bài nhận giá trị từ 1 đến 52, chính vì vậy chúng tôi đưa các con bài lập một dãy theo kiểu **short** trong java. Chúng tôi đã thử nghiệm nén dữ liệu bằng phương pháp tính theo cơ số để đưa dãy các số về một số nhưng nó tỏ ra không hiệu quả do mất thêm thời gian tính toán, trạng thái bị bỏ qua do ra cùng một số sau quá trình nén. Hơn nữa bộ nhớ lưu trữ đủ khả năng để làm việc với dãy kiểu **short** nên chúng tôi chọn phương pháp đơn giản hơn. Để so sánh hai trạng thái, thực chất ta chỉ so sánh hai dãy số, từ đó định nghĩa được phép so sánh hai trạng thái, cái nào lớn hơn, cái nào nhỏ hơn.

2.3.3. Hàm đánh giá trạng thái

Mỗi trạng thái s được đánh giá bởi hàm $\alpha(s)$. Hàm $\alpha(s)$ trả ra một số mà giá trị của nó càng nhỏ thì chứng tỏ trạng thái này càng có nhiều tiềm năng, tức là càng gần với một trạng thái kết thúc nào đó.

Hàm đánh giá $\alpha(s)$ dựa trên các tiêu chí sau:

- Khởi tạo là 0.

- Ở mỗi vị trí đích (foundation), xác định quân bài tiếp theo có thể đặt lên là quân bài nào, tìm quân bài này ở các cột và cộng thêm một lượng giá trị cho hàm đánh giá bằng số lượng quân bài đang che chắn quân bài đó.
- Đếm số lượng các vị trí còn trống (các vị trí free và các cột trống), nếu số lượng các vị trí trống này là 0 thì nhân đôi giá trị hàm vì càng nhiều chỗ trống thì càng có nhiều bước đi có thể thực hiện.
- Ở mỗi vị trí đích (foundation) ta thêm một lượng là số lượng quân bài cần thêm vào để đầy vị trí đích đó vào giá trị hàm đánh giá.

2.4. Truy vết tìm lời giải cuối cùng

Từ trạng thái đích tìm kiếm được, chúng tôi tiến hành truy lại các bước đi đã thực hiện từ trạng thái ban đầu bằng danh sách các bước đi được lưu ở mỗi trạng thái. Mỗi trạng thái sẽ lưu một danh sách các bước đi, biến đổi ngược theo các bước đi này ta sẽ có được trạng thái ban đầu sau quá trình tìm kiếm theo độ sâu K. Thực hiện biến đổi ngược cho đến khi đạt trạng thái ban đầu.

3. Giao diện người dùng (GUI)

3.1. Mô tả giao diện

Hình 4. Giao diện chính

Giao diện chính gồm menu bar và khung hiển thị:

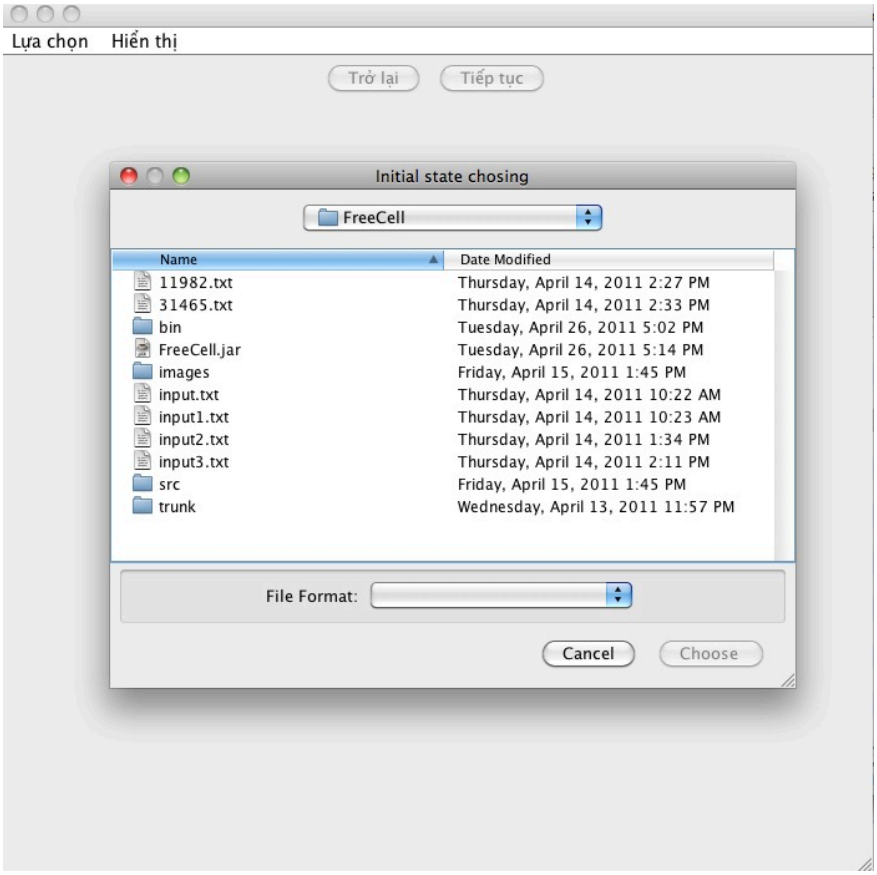
- Menu bar:
  - + File: open (mở chọn file trạng thái ban đầu), exit (kết thúc chương trình)
  - + View: Show movelist (hiển thị danh sách các action mà agent lập kế hoạch để có thể thắng trò chơi, đồng thời cho biết trạng thái hiện tại của môi trường)
- Khung hiển thị:
  - + 2 nút Prev và Next: di chuyển về trạng thái trước đó hoặc tiếp theo bằng cách undo hoặc thực hiện một action ở trạng thái hiện tại.

+ Khung hiển thị: hiển thị trạng thái môi trường (ván bài) hiện tại. Một danh sách các action bên phải cũng sẽ được hiển thị nếu check vào Show Movelist.

3.2. Use case

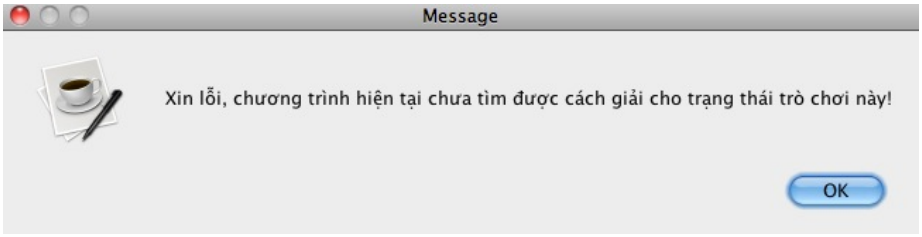
Thể hiện một số trường hợp khi sử dụng chương trình qua GUI:

- Mở File ® Open để chọn file trạng thái ban đầu của ván bài.



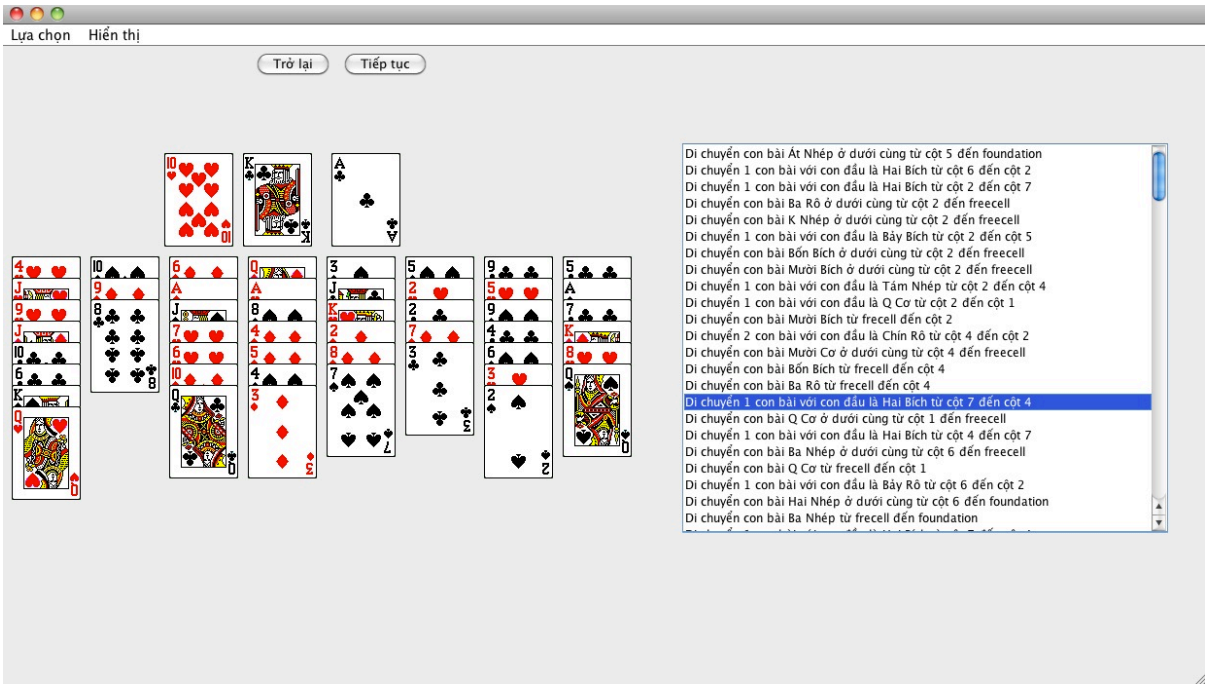
Hình 5. Mở input

- Nếu agent không lập được kế hoạch để đi đến trạng thái cuối cùng (chiến thắng), hộp thoại cảnh báo sẽ hiển thị yêu cầu chọn ván khác.



Hình 6. Không tìm được lời giải

- Nếu agent tìm được lời giải cho bài toán, khung hiển thị sẽ thể hiện trạng thái ván bài bên phải và danh sách các action bên trái (khi Show Movelist được check)



Hình 7. Các bước di chuyển

- Để di chuyển đến trạng thái trước đó hoặc kế tiếp, nhấn nút Prev hoặc Next. Để di chuyển đến trạng thái bất kỳ sau khi thực hiện một loạt các action, lựa chọn trên movelist bên phải khung hiển thị.

3.3. Architecture

GUI trong chương trình sử dụng java swing theo pattern MVC:

- Model: class Card mô hình hóa mỗi quân bài theo rank và suit của nó.
- Control: điều khiển quá trình hiển thị ván bài (môi trường), cụ thể là các trạng thái môi trường qua một số hành động được lựa chọn bởi người dùng.
- View: tải và hiển thị các hình ảnh về quân bài lên khung nhìn.

Người dùng hoàn toàn có thể hiển thị biểu diễn này hoặc ẩn chúng đi. Để xác định các bước của agent, người dùng có thể chọn một move trong list hoặc đơn giản sử dụng 2 nút Next và Pre được dựng sẵn

4. Kết quả thực nghiệm

Chương trình được thử nghiệm trên bộ dữ liệu gồm 32000 trò chơi. Trong số đó: 31700 là tìm ra được lời giải (99.06%), 291 là không có lời giải (0.91%) và 9 là thời gian chạy quá lâu mà vẫn chưa tìm ra lời giải (0.03%).

Ví dụ: Bốn trạng thái trò chơi được đánh giá có độ phức tạp tìm kiếm lớn:
Trò chơi có chỉ số **#31465**:

- Thời gian tìm kiếm: 2.7
- Số lượng bước đi: 149 bước, tính cả các bước thực hiện tự động.
- Số lượng bước ít nhất hiện tại là: 52, chưa tính các bước tự động.

Trò chơi có chỉ số **#29596**:

- Thời gian tìm kiếm: 4.3s
- Số lượng bước đi: 141 bước, tính cả các bước thực hiện tự động.
- Số lượng bước ít nhất hiện tại là: 53, chưa tính các bước tự động.

Trò chơi có chỉ số **#14212**:

- Thời gian tìm kiếm: 2.2s
- Số lượng bước đi: 120 bước, tính cả các bước thực hiện tự động.
- Số lượng bước ít nhất hiện tại là: 52, chưa tính các bước tự động.

Trò chơi có chỉ số **#11982**:

- Sau 8.1s tìm kiếm thì ra không có lời giải.