Các bước xây dựng game caro

1. Giới thiệu  
  
- Trò chơi đối kháng (two-agent,conflicting game http://forums.congdongcviet.com/images/smilies/icon_question.gif) : Gồm 2 người chơi, đối thủ này sẽ tìm cách dành chiến thắng trước đối thủ kia trong một số hữu hạn nước đi, mỗi nước đi đuợc tạo ra dựa từ 1 trạng thái bất kỳ của trận đấu. Nếu sau 1 số giới hạn nước đi, nếu chưa ai dành chiến thắng thì xem như hoà. Ngoài ra, thông tin về trận đấu là hoàn toàn biết đuợc (perfect information) đối với cả 2 đối thủ.   
- Cờ Carô (hay còn gọi là Gomoku ) cũng là 1 loại trò chơi đối kháng, trong đó mỗi đối thủ trong mỗi lượt đi của mình sẽ chọn 1 ô trống còn lại trên bàn cờ (kẻ sẵn các ô lưới ) sao cho tạo thành n con liên tiếp để chiến thắng ... Nếu n = 3 thì nó có 1 tên khác là Tic Tac Toe , nếu bổ sung thêm luật cho nó thì có thể đổi tên là Penta,Pentix (có ăn quân) ... Ngoài ra, có luật thi đấu mà người ta đã chứng minh đuợc người đi truớc bao giờ cung có thuật toán để thắng (thông tin này đáng tin cậy không thì em hông chắc ... chỉ biết là em có đọc qua tài liệu này ...hì hì hì... ), do đó để hạn chế thuận lợi của người đi trước, người ta đã đặt ra "luật rừng" sau ( luật này sẽ sử dụng cho quá trình phát triển chương trình ) :   
+ Bàn cờ có kích thước tuỳ ý NxN, chọn n = 16;   
+ Quân cờ đầu tiên phải đánh chính giữa lưới bàn cờ.   
+ Nếu tồn tại đúng 5 con liên tiếp trên 1 hàng là thắng (chéo,ngang,dọc).[\*]   
+ Nếu hết chỗ đi thì 2 bên hoà.   
+ Và 1 số luật khác, nhưng để đon giản, em dẹp sạch ...   
[\*] : Luật này đáng lẽ là gắt hơn như sau : Đúng 5 con và không bị chặn hai đầu ... nhưng em để dành cho các bác cải tiến ...   
  
2. Thuật ngữ Anh Việt   
  
- Để tiện cho các bác đọc sau này, em xin giới thiệu 1 số thuật ngữ cơ bản sau, dĩ nhiên mớ này là em tự dịch hoặc xem tài liệu nên không tránh đuợc sai sót hoặc chưa chính xác ... mong các bác thông cảm và góp ý ...   
(1) Giới thiệu về không gian tìm kiếm nước đi :   
- Như các bác đã biết, trong trò chơi Caro, cứ sau mỗi nước cờ, mỗi đối thủ sẽ chọn ra từ những ô trống (empty - not occupied) để đi, do đó, sau 1 mỗi nước đi thì số ô trống còn lại sẽ giảm. Như vậy, việc tìm nước đi tiếp theo cho trạng thái có sẵn chỉ là việc tìm kiếm những ô trống còn lại, đồng thời, không gian tìm kiếm sẽ thu hẹp theo số nước đi đã tạo ... Em nói đến điều này là để sau này cải tiến thêm việc gia tăng độ sâu tính toán cho chương trình ở những nước cờ tàn ... Như vậy, để chọn 1 nước đi kế tiếp từ 1 trạng thái bàn cờ có sẵn, ta phải tìm kiếm nước đi ...   
- Không gian chọn nước đi từ mỗi trạng thái ban đầu là hữu hạn, nhưng không gian tìm kiếm (searching space) 1 nước đi dẫn đến chiến thắng là ... hữu hạn luôn (?..hì..?..hì..?..hì... ) ... nhưng rõ ràng số lượng phần tử của hai không gian này đuợc so sánh giống như hạt cát và sa mạc ( hoặc như tập số tự nhiên là vô hạn đếm đuợc, tập số hữu tỉ cũng vô hạn đếm đuợc nhưng mà số lượng phần tử của Q so với của N là 1 trời 1 vực ...) ... Do đó ta không thể vét sạch không gian tìm kiếm nước đi này ( nếu làm đuợc điều này thì làm gì còn những giải cờ nữa ... vì chỉ cần học thuộc thế cờ là xong ...) mà ta phải giới hạn không gian tìm kiếm ...   
- Một không gian tìm kiếm có thể hiện thực theo dạng 1 cái cây đa phân bình thường như trong Data Struct định nghia, lúc này nó đuợc gọi là cây tìm kiếm, cây trò chơi ...( Searching Tree,Game Tree), mỗi nút (Node) cùng mức của cây này thể hiện một lựa chọn các nước đi có sẵn, mức (Ply) này sẽ thể hiện cho việc đánh giá khoảng cách từ nút gốc đến những nút con này ... Nếu số nút ở mỗi mức càng nhiều, tức là có nhiều khả năng chọn lựa 1 nước đi từ 1 trạng thái trước, do đó độ phân nhánh ( Branching factor) của cây này càng lớn ...   
- Dựa vào cái cây trò chơi đã định nghĩa ở trên, việc tìm kiếm nước đi là chọn 1 nút trên cây ( ở mức 1) sao cho nước đó là tốt ( tốt ở đây đuợc hiểu là do mình đánh giá thui, vì 1 nước đi này là tốt hơn nước đi kia thì phụ thuộc trình độ, khả năng của người chơi cờ ...), theo thông thường khi chơi, một nước đi tốt hay không là phụ thuộc vào khả năng dành chiến thắng là cao hay thấp sau khi nước đi này đuợc "made" (tức là "đi"), do đó, muốn chọn 1 nước đi tốt thì nếu chỉ dựa vào thế cờ hiện tại là chưa đủ, mà phải biết thông tin của những thế cờ sau khi chọn nước này để đi ... Ví dụ như khi chơi trò Carô, các bác lại chọn một nước đi vào 1 ô nào đó để chận đuờng 3 hở hai đầu của đối thủ (Opponent, Enemy) vì các bác biết là nếu không đi nuớc này thì sẽ thua ở 2 nửa nước đi tiếp theo, tức là trạng thái thua còn chưa biết đuợc nếu ngay sau khi chọn đi 1 ô khác để đi xuất phát trạng thái này. Khái niệm độ sâu cung nảy sinh từ đây, đon giản thì độ sâu (Depth) là khả năng "nhìn thấy trước" (looking ahead) 1 nước đi tốt sau một loạt nước đi xuất phát từ hiện tại , ví dụ như nếu từ trạng thái này, các bác nhận biết đuợc là sau 6 con nữa là mình sẽ thắng (tức là mỗi bên đi 3 con), khi đó độ sâu tính toán của các bác là >= 6 [not 3], ví dụ này cung chưa chính xác lắm, nhưng 1 ví dụ khác thế này nhá : ở trạng thái hiện tại, các bác chọn đi 1 nuớc đi "tốt" theo sự tính toán của mình, các bác dự đoán là sau 4 con nữa là mình vẫn chưa thua, nhưng thực tế sau đó 3 nuớc nữa (mỗi bên đi 3 con) thì các bác bị thua (..hi hi hi... ), khi đó, rõ ràng "thua" là trạng thái mà các bác không hề nhận thấy khi chọn nước đi tốt ở trên, tức là độ sâu tính toán của các bác trong trường hợp này chính xác là 3, đó cung gọi là độ sâu lớn nhất (Max Depth) mà các bác có thể đạt đuợc khi tìm kiếm ...Như vậy, Max depth thể hiện khả năng & trình độ của người chơi cờ, các bác chơi càng hay thì giá trị này càng lớn ( rõ ràng 1 thằng độ sâu max 6 chơi với 1 thằng độ sâu max 8 thì thằng (6) không bao giờ ăn đuợc thằng (8), hiển nhiên ...)... Khi Max Depth = 0 ==> No thinking ahead   
- Lại nói viết chương trình cho máy tính chơi cờ ... tức là máy tính phải tự tìm nước đi khi mình đưa vào 1 trạng thái bàn cờ bất kì, do không gian tìm kiếm là quá lớn (coi như là vô hạn) nên mình chỉ giới hạn cho máy tính chi tìm kiếm đến 1 độ sâu nào đó mà thôi (cách hiện thực này giống như mình chơi cờ thui ...), đó là độ sâu tìm kiếm lớn nhất ... thể hiện khả năng của chương trình, chúng ta sẽ cố gắng nâng cao giá trị này bằng cách cài đặt thêm các tri thức cờ cho nó (Heuristic, Knowledge ...), tức là sau khi chạy chương trình, máy tính phải biết chọn ra 1 nước tốt trong một mớ các ô trống còn lại sao cho sẽ chưa bị thua sau 1 loạt nước đi (= MAXDEPTH [not MAXDEPTH\*2]) kế tiếp ...   
  
- Một số thuật ngữ[\*] :   
Game Tree : cây trò chơi.   
Searching Tree : Cây tìm kiếm   
Ply : mức của cây (Level)   
Depth : độ sâu (max Ply)   
Iterative Deepening : Tìm kiếm sâu dần   
Quiescence depth : độ sâu tĩnh   
Branching factor : độ phân nhánh   
Parent Node : nút cha   
Children Node : nút con   
Leave Node : nút lá   
Max Node : nút nước đi của đối thủ hiện tại   
Min Node : nút nước đi của đối thủ vừa mới đi   
Evaluation : Sự lượng giá   
Evaluate : Lượng giá   
Static/Dynamic Evaluate : Lượng giá tinh/động   
Lazy Evaluation : Lượng giá lười   
Extension Knowledge - Extension Heuristic : Tri thức bổ sung   
Upperbound : Chận trên   
Lowerbound : Chận dưới   
Make Move : Uýnh 1 nước đi đã chọn   
NULL-Move : nước đi trống   
Move Ordering : Sắp xếp nội nước đi   
Move Generation : Sinh nuớc đi   
Capture : "Đớp"   
Generation Move : Nước đi   
Dangerous Gen-Move : Nuớc đi nguy hiểm (đe doạ)   
Winning thread : Nhánh dẫn đến thắng   
Straight four : 4 con đã bị chận 1 đầu (XOOOO\*, \* là ô trống)   
Opened three : Đuờng 3 mở 2 đầu (\*\*OOO\*\*,X\*OOO\*\*...)   
Broken three : đuờng 3 gãy không bị chận 2 đầu (\*OO\*O\*)   
Winning line : Đuờng thắng (ví dụ : Straight four,Open three,Broken three..)   
Tranpositon Table : Bảng truyền   
Hash Table : Bảng băm   
Offense/Defense - Offensive/Defensive - Aggressive Attack : Các chế độ chiến đấu của chương trình   
Prune, Pruning, Cut Off,...: Cắt nhánh   
Horizontal Effect : Hiệu ứng đuờng chân trời,đuờng ngang   
Back Tracking : Quay lui tìm tới   
Opening Books : Mở Book-Database   
Search by MinMax/AlphaBeta/NegaScout/MTD(f)/PVS Algorithms : Các thuật toán tìm kiếm  
[\*] : Những thuật ngữ trên có 1 mớ đã giới thiệu, mớ còn lại sẽ đuợc giới thiệu ở những bài kế tiếp ...

3. Viết một chương trình Game Gomoku như thế nào   
a. Mục tiêu :   
- Viết 1 chương trình mà máy có khả năng "biết chơi" cờ carô (tức là biết uýnh đúng luật ...hì hì..) ...   
- Chương trình ít nhất phải đánh thắng những người hông biết đánh cờ carô (biết đuợc đuờng thắng...), ngoài ra phải cho những cao thủ người phải "e ngại" vì chương trình có tính người (có suy nghi đàng hoàng)   
- Các khả năng còn lại của chương trình là của các bác ... (dzí dụ : có thể cài đặt để máy "uýnh" dzới máy, "uýnh" nhau trên mạng, "uýnh" multiplayer : 1 người chơi với 2 máy bồ nhau ... hi hi hi ...)   
b. Khởi động[\*] :   
- Chọn cấu trúc dữ liệu thích hợp để lưu trạng thái 1 ô của bàn cờ (lưu thông tin của từng ô)   
- Chọn CTDL thích hợp để lưu trạng thái 1 bàn cờ : Board (dzí dzụ mảng 1 chiều đặc, 2 chiều, danh sách liên kết, cây, chuỗi ...)   
- Chọn phương pháp đánh giá 1 nước đi này là tốt hơn nước đi kia nhu thế nào cho đon giản và hiệu quả (Evaluation)   
- Chọn thuật toán thích hợp để tìm kiếm nước đi ...(No thinking ahead, Simple Breadth First Search, Simple Depth First Search, MinMax, AlphaBeta, NegaScout, MTD(f), PVS ...)   
- Chọn các phương pháp cải tiến để nâng cao khả năng choi cờ (Heuristic : Opening books, Iterative Deepening, Lazy Evaluation, Knowledge Base, Static/Dynamic Evaluation, Winning Threads,Best score, Best move, Deep thought, Quiescence Depth, Transition Table, Hash Table, Score Table ...)   
- Chọn 1 loại ngôn ngữ lập trình để hiện thực (C,C++,C#,Pascal,Object Pascal,Basic,Java...)   
- Tối ưu hoá các chọn lựa trên thành 1 thể thống nhất ...   
c. Phân tích thiết kế giải thuật ...   
d. Hiện thực chương trình ...   
e. Dạy cho chương trình ( Machine Learning): bằng tay, bằng mạng neural ...   
f. Xách gà đi đá : tổ chức một cuộc thi nho nhỏ giữa các chương trình  
[\*] : Ghi chú :   
- Em sẽ trình bày phần a,b,c,d,e. Còn fần f dành cho Admin và các bác Mod nếu đuợc ...   
- Chương trình cơ bản sẽ dùng Alpha Beta để tìm kiếm, các kỹ thuật còn lại sẽ nói sơ wa...   
- Các kĩ thuật ở phần cải tiến nâng cao chương trình cờ, em sẽ chỉ trình bày một số, một số còn lại em sẽ trình bày kỹ thuật, ý tưởng thui !(do em không biết hoặc em nhường lại cho những bác khác Post lên thảo luận ...).   
- Em sẽ dùng C/C++ for DOS (WIN-Console) để giải wuyết vấn đề ...   
- Ở phần e), do trình độ có hạn nên em sẽ luyện cho chương trình bằng tay, còn dùng mạng Neural thì dính tới TTNT nhiều wá mà em chưa được biết rõ, đợi khi nào em hiểu thì sẽ "múa mỏ" ...hic,hic...  
  
Viết chương trình chơi Caro - Phần 2  
1. PHÂN TÍCH VẤN ĐỀ  
  
- Như đã đề cập ở bài trước, để tiện cho việc testing, designing, improvement và optimizing em sẽ sử dụng C++ for DOS để hiện thực, trình biên dịch cụ thể là MS VC++ 6.0. Chương trình sẽ hiện thực trên nhiều lớp ... các bác có thể "thừa kế" để viết cho một số chương trình cờ khác như : Cờ vua(Chess or King Chess),cờ tướng(Chinese Chess)... nói chung việc thiết kế này là sao cho rất hướng đối tượng.  
- Sẽ có thể có những câu hỏi nhỏ đặt ra :  
+ Q1: Tại sao lại là C++?   
==> A: Đơn giản là vì đây là ngôn ngữ lập trình mà em thành thạo nhất (trong số những ngôn ngữ mà em biết ...hic...) và có lẽ đây cũng là ngôn ngữ "đại chúng" nhất (vì hầu như ai cũng học nó như là việc tạo ra cho mình cái nền để học tiếp những ngôn ngữ khác ... ít nhất là sau khi đã biết Pascal!).  
  
+ Q2: Tại sao lại là Object Orient?   
==> A: Vì để tiện cho sự phát triển sau này, chương trình sẽ được phát triển một cách thoải mái và có thể "inherit" từ các lớp cơ bản... Các bác nào nếu chưa wen dzới lập trình hướng đối tượng thì cũng hông seo ... cứ từ từ mà học ... hì hì hì ...  
  
+ Q3: Tại sao C++ for DOS ?   
==> A: Có lẽ câu hỏi này hơi khó trả lời, nhưng mà sẽ dễ dàng nhận ra một điều đó là "Tốt gỗ hơn tốt nước sơn" hay là cụ thể hơn đó là khi cái "ruột" của nó đã không ra gì rồi thì cái vẻ bề ngoài của nó như thế nào thì cũng chỉ được xếp vào hạng "lông" thui chú không được hạng "fly" nữa! Em nói điều này là vì 2 lý do, thứ nhất là hiện nay có nhiều chương trình đánh không ra gì (hic, nói hơi tục là "bèo như cục c.. mèo!") thế mà đã lo làm giao diện này nọ tùm lum,tà la ... không ra cái gì cả!...cái thứ hai đó là cái chương trình Gomoku này của em thiết kế ra chắc đánh cũng không hơn ai, do đó với cái ruột như dzậy rùi thì làm cái vỏ cho đẹp làm chi, răng chừ mừ các bác cải tiến cho nó uýnh như "trâu" thì lúc đó hãy lo làm mấy thứ râu ria cho nó để mà Package-Release ... Tuy nhiên ... hì hì... vì 1 lý do khác là cái giao diện trên Windows đôi lúc làm em và có thể là các bác bối rối... nhưng em hứa sẽ cố gắng khi hết loạt bài này sẽ có cái source for Win với đầy đủ "strength power like a buffalo" cho các bác!... Nói nhỏ nhe : em hay thất hứa lắm...hì hì...  
  
- Kết thúc cho vấn đề này là một lời "nhỏ to tâm sự" : hic ... trong quá trình phân tích và thiết kế sau này( mà có lẽ là ngay dưới đây), có thể những class mà em đưa ra là chưa hợp lý với tôn chỉ "thiết kế hướng đối tượng", khi đó, nếu các bác góp ý thì em sẽ cố gắng xem xét và sửa đổi ... hi vọng là các bác không "refuse" nó ... Nào, nói 1 câu tiếng Anh chơi : "thanh kjù ... du"!...

2. THIẾT KẾ MỘT CHƯƠNG TRÌNH GOMOKU TỔNG QUÁT  
- Bi giờ là đi vào phần phân tích-thiết kế chương trình sao cho nó dzui dzui 1 tí, em sẽ cố gắng trình bày "chi chít" ... ý lộn !... "chi tiết" cách thiết kế từng lớp, có thể mã nguồn chưa tối ưu nhưng chắc là chạy rầm rầm .. hề hề ...việc tối ưu sẽ làm sau khi chương trình đã hoàn chỉnh ...  
- Chương trình sẽ được thiết kế dựa trên các lớp, cơ bản là các lớp như sau : CBaseSearching, CAlphaBeta, CCell, CNode, CBoard, CGomokuInterface, CGomoku ... Cái này thì do quá đơn giản nên hông cần Hierarchy chart ...  
\* Giải thích :  
- Lớp CBaseSearching: đây là lớp tích lũy các tính năng cho một cơ chế searching for moves ( tìm kiếm nước đi)... Đây là 1 base class cho nên chưa đầy đủ các tính năng để đáp ứng cho công việc trên, hầu hết các member là virtual để cho con cháu của nó tự do "múa máy", tuy nhiên nếu đã là con cháu của nó thì ít nhiều gì cũng có đầy đủ các phương thức để tương tác với bạn bè và đồng nghiệp của nó. Các lớp con cháu này có thể thừa kế nó để tạo ra các khả năng (ability) tìm kiếm theo các thuật toán khác nhau... ví dụ các bác có thể thấy một loạt các con cháu của thằng này như là CNoThinkingAhead, CExhaustiveSearch, CMinMax, CMTDf, CAlphaBeta, CNagaScout ...  
- CAlphaBeta: như đã nói ở trên, lớp này kế thừa từ lớp CBaseSearching có nhiệm vụ là tìm kiếm nước đi cho chương trình theo thuật toán tìm kiếm cổ điển là Alpha-Beta Prunning... Vì chương trình này là phiên bản "bèo" nhất nên nó chỉ có khả năng tìm kiếm bằng Alpha-Beta, phần còn lại là của các bác. Nếu sau này muốn cải tiến thì các bác cứ việc kế thừa từ CBaseSearching và thêm một số đặc trưng cho cái searching algorith của các bác (như em đã gợi ý ở trên ...).  
- Lớp CCell là phần hiện thực cho cấu trúc của 1 ô của bàn cờ, có thể bao gồm các thông tin như điểm của mình và đối phương (Score of Friend and Enemy), trạng thái của ô (Status of cell), ở trong cờ Carô thì trạng thái này sẽ là EMPTY,HUM,COM,HEDGE (tức là rỗng, hoặc bị chiếm, hoặc là vùng "bờ giậu", kỹ thuật Hedge sẽ được nói đến khi hiện thực ...). Và 1 câu hỏi đặt ra là tại sao lại chọn cách thiết kế này ? Đó cũng là để phù hợp với tư tưởng "OO" tức là Object Orient, sau này lỡ các bác chuyển wa viết cờ tướng hay cờ vua thì đơn giản là thay đổi thông tin của 1 ô thành những trường khác phù hợp với chính chương trình cờ đó như là SCORES,CAPTURE:(HAS\_BEEN\_CAPTURE, OCCUPIED), STATUS:(EMPTY,ANY\_PIECES\_OF\_HUM or COM or MAN...) rồi xài lại những lớp này.  
- CNode: lớp này rất đáng quan tâm vì nó chính là cái "cốt lõi" trong việc tìm kiếm nước đi, một Node chính là 1 node trong cây tìm kiếm ( Game-Tree, Searching Tree), do đó một Node cũng là 1 nước đi cho nên thông tin của Node sẽ chứa CCell, vì thế CNode có nên kế thừa từ CCell hay không ...? Câu trả lời theo em là không ( mà hông biết giải thích tại sao, hì hì ...) mà nên thêm 1 trường "CCell cell;" trong phần member attribute của CNode ... Cách thiết kế này sẽ tạo nên tính linh động cho cái node, vì nó cũng không nhất thiết phải là một dạng nước đi cụ thể mà đơn giản chỉ là một "node" trong cái cây tìm kiếm với các property khác nhau... Nói thì nói vậy chứ nếu kế thừa từ CCell thì cũng hợp lý ...Chờ sự đánh giá của các bác cho vấn đề này ...  
- Bi giờ là lớp CBoard, một lớp khá wen thuộc nếu thiết kế các trò chơi trên Board (như : các loại cờ, puzzle, brick-game, destroy-ship (bắn tàu) ... và ngay như những trò chơi mang tính "chiến thuật trên từng ô lưới" trên máy super lẫn vi tính như Mộc đế chiến kỳ, Heroes... cũng có thể thiết kế như zdậy, chỉ khác cái Board này nó chứa những đối tượng "animate" chứ hông phải đơn giản là những con số 0,1 như của em mà thui!!!). Thấy em xạo có lợi hại hông ?...hí hí... nói cho vui vậy thôi chứ cái CBoard này nó sẽ chứa mảng các CCell, kích thước mảng chính là kích thước bàn cờ ... dzí dzụ ta sẽ khai báo "CCell board[BOARD\_SIZE];", dzới BOARD\_SIZE là kích thước của bàn cờ carô = 16x16 chẳng hạn. Ngoài ra, lớp này sẽ có thêm những member variables khác như thông tin về trạng thái trò chơi ( nfGameState = GAMEOVER, COMPUTER\_WIN, COMPUTER\_LOSE, RESIGN, HUMAN\_WIN, HUMAN\_LOSE ...) và về trạng thái của lượt đi (nfMoveState = HUM, COM, MAN, STOP...) và về ... đủ thứ nếu thích ...hí hí...  
- Như vậy lớp đảm nhiệm cho việc giao tiếp bên ngoài là CGomokuInterface, vì for DOS do đó chỉ đơn giản là xử lý các công việc : user control (keyboard/mouse), load/write file, link between 2 programs ... ngoài ra update các nước đi cả hai bên lên màn hình, thêm vài thứ lằng nhằng nữa cho nhiều, cho đẹp dzí dzụ thông tin về trận đấu : time ellapse, searching status, some information about last move, thinking status, display some features of the program, for example : version, program's name, programmer's name, current level ... và đặc biệt là những câu gây "shock" cho đối thủ như : "Hê! Đánh dzậy sao ăn tui nổi, chận nè!","Hê...hê, thua rồi ông ơi!","Thấy đường thua chưa?" ... và "Còn 20 nước nữa là thua đó nhe!" ...hic, shock thiệt, em cũng bị shock luôn ... em sợ nó đưa ra câu : "Hic, tha cho em đi, em sắp thua rùi!" thì wê lắm!!!... Hì hì ... nói chung là cái lớp này là lớp console in/out ... nếu sau này bưng nó wa chế độ đồ hoạ thì chỉ cần thiết kế những hàm này cho chế độ đồ hoạ, còn nếu bưng lên Windows thì cũng đơn giản là cho nó con trỏ "CDC \*pDC" là nó làm tuốt ...  
- Lớp tiếp theo là CGomoku: Hic, đây là lớp "be" nhất trong các lớp, có nhiệm vụ "gói"(pack) tất cả các "bé" (other classes) trên vào, đồng thời chỉ đưa ra 1 public method cho người sử dụng là 'RunGomoku()' mà thui, nói như vậy thì bác nào hiểu được chắc cũng ... đã hiểu, còn bác nào chưa hiểu thì em nói tiếp : đó là trong lớp này sẽ thiết kế làm sao cho mấy "em bé" cùng nhau chung sống dzui dzẻ mà không "uýnh nhau" (conflict) như mấy cái hardware của máy em ... ngoài ra nó phải đáp ứng tất cả gì cần thiết như : Initialize, Game-Message-Loop, Ending... một ví dụ đơn giản là nó sẽ xử lý cho user chọn level, thứ tự đi (move order), uýnh nữa hông (continue) ? Cút khỏi chương trình ngay lập tức hông (quit immediately)? Xin thua để uýnh ván mới hông (resign)? Chọn máy uýnh dzới máy coi chơi hông (demo)? Chọn 2 chương trình uýnh dzới nhau hông (tournament: giao tiếp qua file) ?... Nói chung là đủ thứ ...  
- À quên, có 1 lớp nữa em muốn thiết kế thêm đó là : CEvaluate : lớp lượng giá, nhưng suy nghĩ hướng đối tượng của em còn non nớt, không biết wăng vào mấy lớp kia ra sao, nếu bác nào đã từng viết rùi thì share cho em một chút experience ... thực sự là em chưa biết, nếu mà được thì khá hay đó nha : ta có thể có nhiều lớp lượng giá với tha hồ cách lựa chọn (ví dụ : Lazy evaluate, Static evaluate, Dynamic evaluate, Heuristic evaluate với các kỹ thuật như Best score, Best move first, Random-best move ...). Nói thiệt, ý tưởng thì em ...đầy nhưng cách hiện thực thì em ... điếc .. hì hì ... thật sự là mong các bác đã từng viết hoặc đang viết mà có ý tưởng tốt giúp đỡ ...  
- Như vậy, với loạt phân tích kỹ thuật thiết kế trên thì có gì để bàn cãi nữa hông dzậy? Các bác reply in this topic hoặc mail đến em theo địa chỉ [boy4is@hotmail.com](mailto:boy4is@hotmail.com) (địa chỉ[boy4is@fptnet.com.vn](mailto:boy4is@fptnet.com.vn) bữa trước em đưa thì bữa ni nhận hông được nữa, cái này là do server FPT nó lỗi từa lưa... chán wá...á!...). Có bác nào tham gia viết code thì em rất hoan nghênh, hãy giúp em với ... vì nếu tốt sẽ có bài về cờ tướng, cờ vua, cờ ... bất cứ cờ gì mà các bác đưa luật (rule) cho nó ... hic ...xạo đó!... ...  
- Hic, hồi trước giới thiệu các thuật ngữ mà wên giới thiệu luôn các qui ước, bi giờ, để tiện cho sau này thì ... giới thiệu luôn ở cuối bài này chắc cũng chưa quá muộn :  
+ Các Identifier trong chương trình là các từ tiếng "N" (cố gắng như dzậy).  
+ Các tên biến sẽ được đặt theo tiêu chuẩn "hungry" (đói...rét...) tức là hông phải hoàn toàn HUNGARY như mấy bố già Windows Programmers nói, mà sẽ lược bớt còn lại như : kiểu nguyên thì có chữ 'n' ở trước (đôi khi hông có à nhe!), kiểu có cấu trúc thì sẽ ... hông có gì ở trước, kiểu bool thì có chữ 'b' ở trước, ngoài ra có 1 kiểu "đói" riêng em đó là những biến mà xài như một cái "cờ" sẽ có chữ 'f' ở trước (flag # not a function), à wên,kiểu pointer sẽ có chữ 'p' ở trước nhe ... Xài toàn chữ thường (long-hand,normal) cho các biến chỉ có 1 "từ" (word), còn nếu nhiều từ (tức là loại "term") thì sẽ ở dạng Viết Hoa Chữ Đầu (title case).  
+ Các tên lớp thì theo mấy ổng : có chữ 'C' (class) ở trước ... nhưng mà Struct thì hông có từ 'tag' ở đầu đâu nhe (vì đây là chuẩn "hungry" mà!)...  
+ Các tên hàm thì theo kiểu Title-Case như đã nói ở trên.  
+ Các tên hằng thì ...OK...theo tiêu chuẩn ISO 9000..0... là UPPERCASE ALL  
+ Ngoài ra, trong chương trình sẽ có thể có những biến đặt tên rất "ngộ", dzí dụ 'fee'&'ffe'... mà chỉ một mình em hiểu ... hí hí ... đố các bác đó có ý nghĩa gì ? Hè hè ..., đơn giản thui : "Flag-Enemy-Encounter" & "Flag-Friend-Encounter" và nếu gặp ô trống thì sao nhỉ?...Flag-Empty-Encounter == 'fee' ? ==> hì hì, khi đó sẽ là 'fbe' == "Flag Blank Encounter" ... những biến dạng này ý nói khi "đụng hàng" (encounter) là một quân cờ của đối phương (Enemy == Opponent) hay quân ta (Friend == Ally) hay 1 ô trống (Blank == Empty) thì sẽ bật cờ lên rồi xử lý một công việc gì đó ... Xin lỗi, còn nhiều lắm ... nhưng rõ ràng trước với nhau như vậy thì sẽ dễ dàng hơn ... hì hì...  
+ Còn gì nữa hông nhỉ ... chắc hết rồi ...  
- Hêy dà ... còn lại cả mớ bài chắc từ từ wá ... mấy bữa ni phải ôn thi giữa học kì hông thui học lại "phờ râu" à ! Em post lên bài này trong khi ngày mốt,kia,kìa... đã phải thi liên tục rùi ... nói vậy là để mấy bác thông cảm với em, SV mà !!...Tại vì em sợ để lâu quá rồi mà hông nói năng gì thì các bác "refuse" nó luôn nên em phải post lên đó !... Hi vọng sau khi thi giữa kì xong thấy "sung sung" chứ lỡ mà thi "tè le" thì ...hic... Thôi thì muốn biết chương trình tiếp diễn như thế nào ... xem hồi sau sẽ rõ ...

Viết chương trình chơi Caro - Phần 3  
  
Trong phần này, để hiện thực cho những gì đã phân tích ở trên, em sẽ trình bày 1 chương trình cụ thể ... test thử bằng cách đánh dưới chế độ "HUM VS HUM".  
  
- Xin nhắc lại là vì còn trong giai đoạn thử nghiệm nên chương trình của chúng ta chưa thật sự hoàn thành, chỉ có những phương thức cơ bản nhất về giao diện (very simple) và cách thức chơi mới được xây dựng, còn những vấn đề trọng tâm (tức là máy chơi được ...) sẽ được trình bày tiếp và những vấn đề này mới đáng để đi sâu vào ...  
  
- Vì phần xây dựng cho cái vỏ cho chương trình Gomoku này là đơn giản nên em chỉ sơ qua thôi, nếu bác nào thật sự đã lập trình thành thạo C/C++ rồi thì cái source này đối với bác chỉ là "mớ bèo nhèo" thui ... nhưng vì đây là bài viết Tutorial dành cho những bác "tập tễnh" nên em sẽ cố gắng trình bày thật cô đọng, rõ ràng dù vấn đề này là nhỏ ...  
  
- Để tiện cho việc phân tích vấn đề, mời các bác down chương trình ở phần download bên dưới, xin nói lại với các bác lần nữa, vì đây là chương trình rất đơn giản nên còn một số chức năng chưa được include, ngoài ra giao diện rất đơn giản (thậm chí rất "thô" ...) nhưng em hi vọng là các bác hông chê cười (để cho em còn chút "gì đó" mà làm tiếp những bài mới nữa chứ ... hì hì ...). Khi các bác down về nhớ down 2 cái : conio.zip (file thư viện console) và gomoku.zip (file uýnh gomoku) ...  
  
- Em xin trình bày 1 chút về cái chương trình này :  
+ Chương trình viết trên nền DOS với trình biên dịch VC++ 6.0  
  
+ Bao gồm các file hiện thực các lớp : CBaseSearching, CAlphaBeta, CBoard, CNode, CGomoku, CGomokuInterface ... và 1 số file cho các thành phần khác ... Một để ý nhỏ đó là em đã dẹp cái lớp CCell sang một bên vì sau một hồi suy ngẫm, em thấy nó không cần thiết, nếu không có nó cũng không làm mất tính "Orient Object" của chương trình, mặt khác nó đỡ làm rối rắm hơn (bởi vì đơn giản là mình chỉ cần coi 1 cell là một biến int mà thôi ... phần còn lại là do 1 node lưu trữ ...).  
  
+ Lớp CBaseSearching gồm có 1 số thành viên tiêu biểu như Quiescence() (tìm kiếm độ yên tĩnh bằng cách dùng AlphaBeta algorith), m\_nFixDepth (độ sâu cố định), m\_nQuietDepth (độ sâu tĩnh), Eval() (lượng giá nước đi ...). Ngoài ra, trong quá trình phát triển sẽ có thể nảy sinh thêm nhiều hàm (biến) thành viên khác (ví dụ hiện thực cho phần NULL-move, Hash Table, Transition Table ...).  
  
+ Lớp CAlphaBeta kế thừa public từ CBaseSearching có những hàm thành viên tiêu biểu như AlphaBeta() (tìm kiếm nước đi với thuật toán AlphaBeta), Search() (call AlphaBeta()), Eval() (hàm lượng giá cụ thể cho Gomoku). Lớp này cũng có thể thay đổi trong quá trình phát triển chương trình.  
  
+ Lớp CNode đơn giản chỉ có 2 hàm tạo, 2 biến thành viên nState (trạng thái của ô trên bàn cờ : HUM,COM,EMPTY...), nMove (thứ tự ô trên bàn cờ)... Sau này khi xây dựng những hàm lượng giá thì sẽ thêm vào những phương thức, biến mới ...  
  
+ Lớp CBoard : đây là lớp có khá nhiều phương thức nhằm tương tác với cái Gomoku Board như : Gen() (sinh những nước đi bình thường), GenDanger() (sinh những nước đi nguy hiểm như đường 3 mở, đường 4 ... dùng trong Quiescence() ), MakeMove() (thử nước đi), Restore() (phục hồi lại trạng thái cũ của Board sau khi backtracking), LegalMove() (xem thử một nước đi có là hợp lệ hay không ... tức là kiểm tra ô đó còn trống hay không ...), SortMoveGen() (sắp xếp nội các nước đi hợp lệ để hi vọng trong quá trình search thì AlphaBeta có thể cắt nhánh nhanh hơn), CheckFiveInRow() (xem đã đủ 5 con trên 1 hàng ngang,chéo hay dọc chưa ...). Ngoài ra có 1 số biến thành viên đó là : nMoveState (lượt của người đi nước hiện tại), nHistStack (lưu lại các nước đi đã thử để backtracking), nMoveStack (lưu lại các nước đi sinh ra bởi hàm Gen() hay GenDanger() ), nTopMoveStack (đỉnh của nMoveStack chứa số lượng tối đa các nước đi đã sinh đến 1 độ sâu nào đó ...), nTopHistStack (đỉnh của nHistStack lưu lại số nước đi đã thử : make move). Biến nMask là dùng làm mặt nạ để cho việc sinh và kiểm tra các nước đi trùng nhau nhanh hơn ...  
  
+ Lớp CGomokuInterface là đơn giản, vì đây chỉ là lớp đảm nhiệm việc nhập xuất ra console nên các hàm và biến thiết kế là để cho tiện công việc này. Để ý hàm GetMessage() : tuy chưa làm xong nhưng ý tưởng là dựa trên điểm số đạt được và trạng thái của bàn cờ sau khi computer search để convert những chỉ số này (lưu tại biến nMessageIndex) thành những reference-index cụ thể cho việc in ra những messages gây shock cho đối thủ !! Hì hì, cái này sau này sẽ tính ...  
  
+ Lớp CGomoku thì cũng đơn giản, chủ yếu là liên kết các lớp trên lại để tạo thành 1 chương trình hoàn chỉnh, chạy được ... Phương thức RunGomku() sẽ cho làm 1 vòng lặp : "Khởi đầu --> Đánh nhau --> Kết thúc ván cờ --> Thoát hay uýnh nữa thì lặp lại", trong vòng lặp "đánh nhau" thì mỗi bên đi một nước, nếu có người lập được "Five in a row" thì sẽ qua bước "Kết thúc ván cờ". Lớp này cũng coi như là chưa hoàn chỉnh vì chưa có đầy đủ chức năng như Opening Book, Load/Save from file (Serialize() ) ... rồi sẽ include sau ...  
  
- Xong rùi, bi giờ là F7 ==> Ctrl F5 ==> Enjoin it !!! À quên, khi biên dịch nhớ gửi cái file console library vào thư mục gốc của nó nhe !! ( Cái này thì các bác phải dịch consoleIO sang lib trước đã .. nó sẽ tạo 2 file : RConIO.lib (Release), DConIO.lib (Debug)...).  
( Bài 4 : BƯỚC VÀO CÁC KỸ THUẬT SINH NƯỚC ĐI )

Viết chương trình chơi Caro - Phần 4  
  
1. Giới thiệu các kỹ thuật:  
  
- Đến lúc này thì các bác đã có chương trình Gomoku trong tay rồi, "muốn chém muốn giết thì tuỳ ý" ... hic ... do đó cũng đã đến lúc bàn đến những kỹ thuật khác trọng tâm hơn trong cái chương trình carô này !! Các kỹ thuật này sẽ được trình bày bao gồm : Sinh nước đi (Generate moves), Lượng giá (Evaluate, Evaluation), Tìm kiếm nước đi (Search game tree), Bổ sung tri thức cờ (Heuristic), Lập nước đi có sẵn (Book Opening, Book database), Luyện chương trình ( Machine learning)... Mỗi kỹ thuật trên lại có những kỹ thuật "con con" nữa như sau :  
+ Generate moves : full generation, increasing generation, decrescent generation, complete generation, selective generation ...  
+ Evaluate : static, dynamic, some technics ...  
+ Search game tree for move : Minimax/AlphaBeta/MTD(f)/NagaScout pruning, Quiescence , Ordering moves, Interative Deepening, Hash table, NULL-Move, Endgame Database ...  
+ Heuristic : Evaluate, Winning lines, Offensive/Defensive mode, Winning threads, Related cell ...  
+ Book opening : save book-format, read book, organize database ...  
+ Machine learning : by handle ( modify score-table )...  
+ Bài đầu tiên của các loạt kỹ thuật này sẽ là SINH NƯỚC ĐI và cách tổ chức cấu trúc dữ liệu sao cho hiệu quả về tốc độ ...  
  
2. Kỹ thuật sinh nước đi:  
\* Một số câu hỏi :  
Q1 : Sinh nước đi là gì ?  
A1 :   
- Sinh nước đi (generate moves) đây là 1 giai đoạn trong quá trình tìm kiếm nước đi, tức là từ trạng thái hiện tại, mình muốn tìm kiếm một nước đi hợp lệ cho lượt của người chơi hiện tại ==> khi đó phải lượm (tức là "sinh") tất cả các nước đi có thể có, sau đó mình sẽ chọn một nước đi tốt nhất (phải đánh giá nó ...) trong số những nước đó.  
Q2 : Tại sao phải sinh nước đi ?  
A2 :  
- Thông tin cho quá trình tìm kiếm nước đi chỉ đơn giản chỉ là cái board với những ô trống hay ô đã bị chiếm (của cả 2 bên) mà thôi! Nếu không có sẵn thông tin những nước đi hợp lệ thì việc tìm kiếm đơn giản chỉ là chọn đại 1 ô trống rồi uýnh 1 con cờ ... Rõ ràng việc làm này sẽ dẫn đến việc tìm kiếm vô cùng phức tạp vì mình sẽ không biết 1 ô trống đã search qua hay chưa ==> do đó có thể rơi vào 1 vòng lặp tìm kiếm vô hạn ... Ví dụ cụ thể là giả sử mình đang ở trong 1 mê cung, bước đầu tiên là chọn 1 con đường không bị tường chặn rồi đi vào, nếu đụng ngõ cụt thì sẽ quay lại tìm đường khác, thế nhưng nếu sau khi quay lại, mình lại không biết là đã từng vào con đường trên và thế là đi vào tiếp và ... Cách giải quyết cho trường hợp này đó là trước khi chọn 1 trong những con đường để đi thì mình sẽ đánh dấu tất cả con đường theo thứ tự ... và lần này bước vào con đường thứ 1, nếu không thoát ra được thì quay lại và chọn đường thứ 2 ...  
==> Nói chung phải sinh nước đi để có thể duyệt chính xác và đầy đủ ...  
  
\* Phân tích các kỹ thuật :  
[a] Full generation : sinh đầy đủ các nước đi  
- Ý tưởng : Từ trạng thái hiện tại, kiểm tra xem còn bao nhiêu nước đi hợp lệ có thể có thì cất tất cả vào stack ...  
- Phân tích :  
+ Mã giả như sau :  
for (i = 0; i < BOARD\_SIZE;i++)  
if (IsLegalMove(i)) // kiểm tra xem (Board[i] == EMPTY)  
if (IsNotInGenMove(i)) // kiểm tra xem đã có trong danh sách chưa...  
StoreGenMove(i) // cất vào stack ...  
+ Trong các loại board-game thì kỹ thuật này chỉ ứng dụng khi việc sinh nước đi từ trạng thái có sẵn là đơn giản, ngoài ra việc đánh giá nước đi này là hay hơn nước đi khác phải làm thật chính xác vì nếu không sẽ khó thấy được nước đi nào là tốt hơn nước kia trong hàng đống nước đi đã sinh. Một số tài liệu nước ngoài phân tích về vấn đề này có nói rằng : "Nếu bạn có thể kiểm tra mọi nước đi khá nhanh thì không có vấn đề gì cho việc sinh toàn bộ các nước đi có thể có ... bởi vì bạn sẽ chọn được nước đi thật sự tốt và việc sinh nước đi và tìm kiếm nên được làm càng nhanh càng tốt!", quan điểm này theo em là đúng mà không đúng, đúng là vì "...việc sinh nước đi và tìm kiếm nên được làm càng nhanh càng tốt!..." và luôn tìm được 1 nước đi tốt nhất ... nhưng không đúng là ở chỗ "... không có vấn đề gì cho việc sinh toàn bộ các nước đi có thể có ..." bởi vì Gomoku là trò chơi có độ phân nhánh rất cao (tức là số cách chọn nước đi ứng với 1 trạng thái có sẵn), không những vậy, trong trò Gomoku thì rất nhiều nước đi nếu không có quan hệ với những nước có sẵn thì sẽ dư thừa và không được chọn ... Và một điểm khác so với những board-game khác đó là việc sinh nước đi trong Gomoku là khá nhanh và rất dễ ... Do đó túm lại là kỹ thuật này nếu ứng dụng trong Gomoku-Game sẽ bị phá sản ... Em trình bày ra ở đây là dành cho những bác viết các loại cờ khác tham khảo ... bởi vì, ví dụ trong cờ tướng, đôi khi một nước đi hơi ngớ ngẩn như lên "chuột"(tốt,chốt : pawn) cũng có thể chống lại một bàn thua trông thấy ...  
[b] Increasing generation : Sinh dần các nước đi ...  
- Ý tưởng : Từ trạng thái ban đầu, sinh 1 vài nước đi "nhạy cảm" (có triển vọng đưa đến chiến thắng), kiểm tra xem những nước đi này sau một vài độ sâu kế tiếp xem có thật sự là "cúm" chưa ? Nếu chưa thì tiếp tục sinh những nước đi khác rồi kiểm tra tiếp ...  
- Phân tích :  
+ Mã giả như sau :  
\* Search : Generate move-gen (step 0)  
for (i = 0;i < BOARD\_SIZE;i++)  
if (IsLegalMove(i))  
if (IsHasOutlook(i)) // có triển vọng ...  
if (IsNotInGenMove(i))  
StoreGenMove(i)  
\* Make a move in stack ...  
\* Increase depth and repeat until deep enough ==> Cut off ?? ==> Ohh, no...  
\* Trở lại bước 0  
+ Kỹ thuật này khá hay, nhất là khi mình muốn tính sâu vào những nước "tốt" (theo quan điểm của mình) để mong chờ search-algorithm cắt nhánh sớm trước khi mình tiếp tục sinh thêm những nước lằng nhằng khác ... Tuy nhiên, nếu ở 1 trạng thái nào đó, nước đi tốt không nằm ở trong số những nước đi có triển vọng thì mình vẫn phải tiếp tục sinh ... cách cài đặt sẽ trở nên phức tạp. Rõ ràng cách này chỉ nên sử dụng để đối phó trong trường hợp bị giới hạn thời gian, tức là mình sẽ lựa 1 nước đi tốt nhất trong số đó "để dành", nếu thời gian không đủ để generate && search tiếp thì mình sẽ "moi" nó ra mà uýnh "đỡ" ... Một nhược điểm khác đó là chưa chắc tìm thấy một nước đi tốt nhất như kỹ thuật đầu tiên (vì nó đã lỡ cắt nhánh ở 1 nước đi mà tưởng rằng là tốt nhất, hiện tượng này cũng có thể gọi là "cực đại cục bộ") ...  
[c] Decrescent generation : Sinh nước đi giảm dần  
- Ý tưởng : Cách này đọc qua tưởng chừng như ngược lại với cách trên nhưng thực sự là có sự khác biệt rất lớn ! Nói chung là chọn một thang đo ban đầu để đánh giá tĩnh sự lợi hại của các nước đi hiện tại cần sinh ... Sau đó lần lượt giảm thang đo và sinh nước đi cho đến khi nào chứa ít nhất 1 nước đi trong stack thì break ... nếu chưa thì sẽ tiếp tục sinh tiếp ...  
- Phân tích :  
+ Mã giả như sau :  
\* ExistAtLeastOneMoveGen = FALSE;  
\* for (mark = HIGHEST;;--mark) // đi từ mức đánh giá cao nhất giảm dần  
for ( i = 0;i < BOARD\_SIZE;i--)  
if (IsLegalMove(i))  
if (ValueIt(i) == WORTH) // nếu thật sự có giá trị  
if (IsNotInGenMove(i))  
StoreGenMove(i)  
ExistAtLeastOneMoveGen = TRUE;  
if (ExistAtLeastOenMoveGen) break; // else continue;  
+ Rõ ràng cách này luôn tìm ra nước đi để đưa vào move-stack từ đó thể hiện sự đúng đắn của nó ... Kỹ thuật này khá phổ biến trong các chương trình cờ (ít nhất là trong số những chương trình mà em biết !!) bởi vì nó sẽ lần lượt chọn lựa những nước đi từ tốt đến xấu nhất dựa trên thang đo là giá trị của mark và nó bảo đảm sẽ sinh & chọn ra những nước tốt nhất có thể từ trạng thái hiện tại để đưa vào stack. Nói là vậy nhưng rõ ràng điều này có hiện thực được hay không là còn nhờ vào hàm ValueIt() nữa! Nếu hàm này làm nhanh và chính xác thì sẽ đem hiệu quả cao ... cách thiết kế hàm này cũng tương tự như hàm IsHasOutlook() ở trên nhưng cần rộng và chính xác hơn tức là phải tính thêm những nước được xem là "outlook" đối với đối thủ nữa, bởi vì những nước đó có thể thật sự có giá trị (do nếu ta đánh vào nước đó sẽ chiếm vị trí "chiến lược" của đối phương, tức là chận trước những "đường mở" của đối thủ ...và đôi khi việc làm này là đáng giá hơn việc đánh 1 đường 5 (thuộc loại có triển vọng của ta) mà đối thủ dễ dàng chận phá !!). Kỹ thuật này thật sự rất hay nếu nó "thiếu vắng" có những nhược điểm sau, đó là việc thiết kế hàm ValueIt() thật sự là rất khó, bởi vì mình không thể lập trình chính xác cho máy biết được nước nào là có giá trị, đây chỉ là gần đúng mà thui, hic ... với lại khi thiết kế xong rồi mà bưng nó vào cái hàm search nước đi của mình thì chắc sinh ra được 1 nước đi phải mất hàng đống micrô giây quá (là quá lớn so với chu kỳ clock của CPU rồi đó!!!)... Do đó phải cân nhắ

d] Complete Generation : Sinh tất cả các nước đi  
- Ý tưởng : Sử dụng stack để lưu tất cả nước đi của trạng thái bắt đầu seach, sau đó việc tìm kiếm nước đi sẽ không cần sinh thêm lần nào nữa ... Tức là chỉ cần sinh 1 lần và cất ngay vào move-stack, sau này khi lấy ra sẽ kiểm tra xem nó có còn là 1 legal move hay không (tức là ô này đã bị đánh chưa ...) rồi analyze nó...  
- Phân tích :  
+ Mã giả cho chương trình thì tương tự như kỹ thuật Full generation  
+ Cách này sẽ không tốn thời gian sinh nước đi trong quá trình search, không tốn nhiều bộ nhớ để lưu các biến nước đi theo từng độ sâu nhưng tốn nhiều thời gian cho việc phân tích và chọn lựa nước đi. Có những đặc điểm trên thì dường như nó tiến bộ hơn Full generation nhiều !... tuy nhiên đó là trong trường hợp không kèm thêm việc lượng giá nước đi cùng với việc sinh nước đi (mà cái này lại hay xài)... bởi vì việc tìm kiếm chọn lọc nước đi là phải dựa trên chỉ số đánh giá nào, chỉ số đó được tính ra sao và khi nào, ở đâu ... Nói chung đây cũng là kỹ thuật chỉ ứng dụng cho từng trường hợp và từng trò chơi cụ thể !!...  
[e] Selective generation : Sinh nước đi có chọn lọc  
- Ý tưởng : Dùng một hàm chọn lựa để đánh giá những nước đi nào là đạt những tiêu chuẩn đặt ra thì mới đưa vào stack, rõ ràng hàm chọn lựa này là rất rộng, ví dụ : chỉ sinh những ô trong vòng bán kính 1,2 ô so với những ô đã đánh, chỉ sinh những ô mà có thể gây nên những đường đe doạ, chỉ sinh những ô nằm trong khu vực "nóng", chỉ sinh những ô mà có thể chận được đường hiện tại ... nói chung là rất nhiều tiêu chuẩn ... kỹ thuật này thì tổng quát hơn những kỹ thuật sinh có chọn lọc ở trên.  
- Phân tích :  
+ Mã giả như sau :  
while (ExistMoveGenSatisfiesTheCriteria())  
if (IsNotInGenMove(GetGenMove())  
StoreGenMove(GetGenMove())  
+ Thấy mã giả khá đơn giản nhưng thật ra là phức tạp vô cùng, ngay từ việc thiết kế những hàm đo tiêu chuẩn đã khó rồi, huống chi là tổ hợp của tất cả hàm đó lại ... tuy nhiên cách này sẽ mang lại hiệu quả khá cao dựa trên qui luật bù trừ (ví dụ nếu search nhanh thì mình sinh ra nhiều move-gen và đánh giá chúng sơ sơ, search chậm thì đưa ra ít move-gen nhưng mỗi nước đi đều đã qua bước chọn lọc kỹ càng, chỉ cần "móc" 1 nước đi ra là đã có ngay một nước đi khá tốt rồi! ). Ngoài ra nếu phối hợp cách này với những kỹ thuật ở trên thì càng tuyệt, ví dụ đơn giản là mình chỉ sinh những nước đi trong vòng bán kính 2 ô so với những ô đã đi và lựa những nước đi thoả thang đo nào đó ( nếu đang ở mode offensive thì sẽ sinh những nước gây ra đường đe doạ còn ngược lại đang ở mode defensive thì lượm những nước có triển vọng của đối phương mà uýnh... ), khi đó, số lượng những nước đi thoả các tiêu chuẩn trên sẽ không nhiều (tức là độ fân nhánh thấp) mà chất lượng của nước đi sẽ được nâng cao (do toàn lựa những nước đi có quan hệ với những nước đi đã uýnh + bảo đảm nếu khi attack thì đối thủ chận túi bụi còn khi đang defend thì đối thủ ngồi khóc ròng vì hông còn nước nào để phát triển đường ...), khi đó ta còn có thể gia tăng độ sâu (cố định lẫn tĩnh) lên thêm vài nấc, nên nhớ rằng việc gia tăng độ sâu làm cho chương trình sẽ có tính người hơn và uýnh hay như "trâu", các bác thử nghĩ nếu đang đánh với 1 chương trình cờ mà nó đưa ra thông báo là 'còn hai ba chục nước nữa là nhà ngươi sẽ thua' thì có bị shock hông chứ, em thì chưa thấy chương trình nào như vậy cả, hề hề !!!. Hic, vì sự lợi hại của kỹ thuật này, chương trình Gomoku của em sẽ tiến triển theo hướng này và phối hợp thêm một số kỹ thuật khác ở trên.  
  
>> Như vậy là xong phần Generate-move technics \*\*  
  
\* Phân tích xong các kỹ thuật, em cũng xin giới thiệu lại 1 vài thuật ngữ hoặc tên hàm em chế ra và xài trong quá trình sinh nước đi để cho các bác tiện theo dõi :  
+ Generate move : sinh nước đi.  
+ Gen/GenDanger : các hàm sinh nước đi bình thường hay đe doạ.  
+ Move-Gen : nước đi được sinh.  
+ Increasing/Decrescent generation (Increment/Decrement) : Sự gia tăng hay giảm bớt trong quá trình sinh nước đi.  
+ Offensive/Defensive move : Nước đi tấn công hay phòng thủ ...  
+ Aggressive/Offensive/Defensive mode : các chế độ tấn công hùng hổ hay phòng thủ chặt...  
+ Complete/Selective generation : Sinh toàn bộ nước đi hay là sinh những nước đi có chọn lựa.  
+ Forward pruning : những nước đi được xem như là 1 sự cắt nhánh trước khi search.  
+ Store Generated Move : cất nước đi đã được sinh vào stack  
+ Is Not In Generated moves : chưa có trong mảng (stack) lưu các nước đi đã sinh...  
  
**3. Viết mã cho hàm sinh nước đi:**  
  
- Số nước đi cần sinh trong mỗi bước đi là rất nhiều ( = Tổng số ô trên bàn cờ - Số ô đã đi) nhưng chỉ có 1 ít trong số đó là có liên quan với những ô đi trước và có khả năng gây đe doạ ==> đó là những ô kế cận những ô đã đánh ... ở đây có thể sinh tất cả những nước đi trong vòng bán kính 1,2 ô so với ô trung tâm ... Ví dụ hình sau cho thấy những ô liên quan cần sinh trong vòng bán kính là 1,2 đối với quân 'O' ở trung tâm :  
  
-\*-\*-\*- -------  
--\*\*\*-- --\*\*\*--  
-\*\*0\*\*- --\*O\*--  
--\*\*\*-- --\*\*\*--  
-\*-\*-\*- -------  
R = 2 R = 1  
Áp dụng phương pháp này và sinh tất cả các nước đi cho mọi ô đã có quân chiếm, với cách sinh nước đi hạn chế này thì sẽ giảm rất nhiều chi phí cho việc tìm kiếm nước đi (vì mỗi lần duyệt ở độ sâu mới thì chỉ cần lựa ra một số nhỏ hơn nhiều so với số ô trống còn lại) ==> dùng nó để tăng thêm độ sâu.  
- Việc sinh nước đi theo kiểu này có thể để riêng trong 1 hàm hoặc các bác có thể cải tiến bằng cách vừa lượng giá vừa sinh nước đi (sẽ nói rõ trong phần lượng giá), chương trình em thì để riêng trong hàm PushMoveGen().  
- Sinh nước đi phụ thuộc nhiều vào CTDL lưu bàn cờ : mảng 1 chiều, mảng 2 chiều, danh sách ..., mỗi cách lưu thì sẽ có cách sinh nước đi khác nhau, có ưu và khuyết riêng, em thì chọn cách dùng mảng 1 chiều để lưu bàn cờ nên cách sinh nước đi cũng dễ dàng hơn như sau :  
+ Kích thước bàn cờ cần lưu : BOARD\_SIZE = EXSIZE\*EXSIZE = (SIZE+2)\*(SIZE+2)  
+ Lưu Board : int board[BOARD\_SIZE];  
+ Lưu mảng mặt nạ : int nMask[BOARD\_SIZE];  
=> Như vậy từ vị trí 'nMove' trên bàn cờ, có thể nhảy đến 8 vị trí xung quanh bằng cách sử dụng mảng DIRECTION mở rộng như sau (giả sử quân X đang ở ô thứ nMove trên board)

const int EXDIRECTION[9] =  
{EXSIZE,EXSIZE-1,1,EXSIZE+1,-EXSIZE,-EXSIZE+1,-1,-EXSIZE-1,0};  
  
  
- - - - - - - nMove + EXDIRECTION[3]  
- - - - - - - ==> Truy cập đến ô số 3  
- - 7 4 5 - - nMove + EXDIRECTION[4]  
- - 6 X 2 9 - ==> Truy cập đến ô số 4  
- - 1 0 3 - - nMove + EXDIRECTION[3]+EXDIRECTION[3]  
- - - - - 8 - ==> Truy cập đến ô số 8  
- - - - - - - nMove + EXDIRECTION[3]+EXDIRECTION[5]  
=> Truy cập đến ô số 9  
=> Từ đây mình cải tiến việc truy cập đến những ô trong vòng bán kính 2 ô so với ô trung tâm như sau :  
  
  
const int EXDIRECTION[17] =  
// 8 ô xung quanh (bán kính = 1)  
{EXSIZE,EXSIZE-1,1,EXSIZE+1,-EXSIZE,-EXSIZE+1,-1,-EXSIZE-1,  
// 8 ô tiếp xung quanh (bán kính = 2)  
2\*EXSIZE,2\*EXSIZE-2,2,2\*EXSIZE+2,-2\*EXSIZE,-2\*EXSIZE+2,-2,-2\*EXSIZE-2,  
// Đánh dấu kết thúc số hướng mở rộng  
0};  
  
  
Chương trình em thì chỉ có truy cập đến 8 ô (R = 1) xung quanh thui, do đó dùng mảng EXDIRECTION[9] ở trên ... Có nhiều bác cho chương trình mình tính luôn R = 2 nên chương trình tính nhiều hơn mà đánh hông hay lắm, một bằng chứng là chương trình thằng bạn em đánh 'hết xảy' mà chỉ cần tính 8 ô xung quanh thui, bác nào thích thử sức với chương trình của mình (hoặc là với các bác) thì xin mời download bên dưới, chương trình này thằng bạn em viết lâu rùi nhưng hiện giờ vẫn chưa thấy đối thủ (hì hì, trong nước và một đống của nước ngoài) (chạy trên DOS và tiếc là hông có source vì Hard disk của nó phải 'fo mát đầu bò' rùi!).  
- Ở đây dùng mảng kích thước BOARD\_SIZE = EXSIZE\*EXSIZE chứ không dùng mảng BOARD\_SIZE = SIZE\*SIZE là vì để giảm bớt việc kiểm tra các nước đi vượt ra ngoài bàn cờ (bằng cách nới rộng board thêm 1 cho mỗi biên), dùng mảng nMask là để kiểm tra một nước đã vượt biên hay chưa, hoặc là nước đó đã nằm trong danh sách sinh nước đi hay chưa ... Nếu bác nào đã từng đọc qua PC World viết về chương trình VSCCP thì sẽ thấy cách cài đặt của chương trình em là tương tự theo cấu trúc đó, em đã viết hàm sinh nước đi PushMoveGen(CNode &move) tại độ sâu nPly cho ô trung tâm move (CNode) như sau :  
  
  
0 int i;  
1 char Step;  
2 const int \*pDirect = EXDIRECTION;  
3 while((Step = \*pDirect++) != 0)  
4 {  
5 i = move.nMove + Step;  
6 if (!nMask[i])  
7 {  
8 nMask[i] = 1;  
9 MoveStack[nTopMoveStack[nPly]] = i;  
10 nTopMoveStack[nPly]++;  
0 }  
0 }  
  
  
  
==> Dòng 2 và 5 : Nếu không sử dụng mảng EXDIRECTION thì sẽ khó mà truy cập đến vị trí của những ô xung quanh (TopLeft,TopRight,Bottom,Top,BottomRight,BottomLef t,Left,Right)  
==> Dòng 5 và 6 : Nếu không sử dụng mảng kích thước EXSIZE\*EXSIZE thì có lẽ phải thêm dòng kiểm tra đã vượt biên hay chưa, nếu không sẽ truy cập "lộn tiệm" ...  
==> Dòng 6 : Nếu không sử dụng mảng nMask (= 0 nếu là ô trống chưa sinh, = 1 nếu là đã có hoặc là đã vượt biên) thì chắc cũng phải dùng 1 hàm tìm kiếm tuyến tính xem nước đi i = move.nMove + Step đã có trong danh sách hay chưa ...  
==> Mấy dòng kia là cập nhật nước đi trong MoveStack thui (mang tên là MoveStack nhưng nó cài đặt trên mảng hoàn toàn !)  
- Nếu cài đặt cho việc sinh các nước đi có thoả mãn các điều kiện nào đó không thì thay dòng 6 tương tự như sau :  
  
  
if (!nMask[i]&&IsSatisfyAnyCriteria(i))  
{  
...  
}  
  
  
Tức là kiểm tra thêm một số tiêu chuẩn 'chọn lọc' nước đi mà mình đề nghị ... để giảm thiểu số lượng nước đi ... Ví dụ sau này chúng ta sẽ cài đặt IsSatisfyAnyCriteria() là chỉ sinh những nước nguy hiểm thật sự như đường 3,đường 4... trong quá trình tìm kiếm độ sâu yên tĩnh.  
- Bây giờ sinh nước đi cho tất cả thì có thể đơn giản như sau :  
  
  
for (int i = 0;i < nTopHistStack;i++)  
PushMoveGen(HistStack[i].nMove);  
  
  
Trong đó nTopHistStack là số quân cờ đã đánh, HistStack là mảng CNode lưu tất cả vị trí theo thứ tự đã đánh ... Sau này muốn thêm mục Undo thì có mảng này rồi làm cũng dễ, muốn RePlay lại thì cũng chỉ cần 'tua' lại theo thứ tự đã lưu trong mảng History này rồi !...  
- Còn gì đáng nói trong phần này nữa không nhỉ ?? Thui thì em qua phần mới : EVALUATE  
  
EVALUTE như thế nào?  
  
1. Bác nào chơi Cờ cá ngựa thì tính nước đi như thế nào ?  
2. Bác nào chơi Cờ tướng thì bác sẽ tính 1 nước đi cho mình là như thế nào ?  
3. Bác nào chơi Caro thì tính nước đi như thế nào ?  
  
Câu 1 : Khỏi trả lời ...  
Câu 2 : Mời bác trả lời ...  
Câu 3 : Xin bác 'se' em chút kinh nghiệm của bác nào ...  
- Thui thì cho em nói, có gì sai các bác sửa http://forums.congdongcviet.com/images/smilies/biggrin.gif ... Cờ tướng muốn tính nước đi hay thì trước hết phải am hiểu luật cờ để từ đó biết được giá trị của các quân cờ, theo em, giá trị quân cờ trong cờ tướng là phải xét hàng đầu ... tiếp theo là thế cờ , nhớ lại rằng : nếu hông có quân cờ cơ động (tức là giá trị càng cao) thì liệu có thế cờ tốt hay không và nếu có nhiều quân cờ 'xịn' hơn đối thủ thì cơ hội chiến thắng sẽ càng cao, tuy nhiên việc đánh giá này là động (dynamic) vì đôi khi vì để tướng hông bị 'cạp' thì 2 xe cũng phải hi sinh, còn về thế cờ, tức là chiến thuật trong cờ tướng thì phải nói là phức tạp vô cùng, nhất là vì mình phải viết cho máy chứ máy nó hông tự học từ kinh nghiệm qua trận mạc như mình được (dĩ nhiên nếu có cài máy học qua trận mạc thì cũng chỉ 1 phần thui, đâu ứng dụng được nhiều, theo em thì DeepBlue nếu vác đi đánh với ông khác chứ không phải là Kasparow thì có lẽ nó cũng thất bại ê chề bởi vì kinh nghiệm mà nó học được là của người ta dạy chỉ để đối phó với ổng Kasparow thui...), do đó thường mình chỉ cài khai cuộc cho nó vì đơn giản là đọc book khi số nước đi chưa nhiều, còn khi đã bước qua trung cuộc rồi thì chỉ cài vài cái thế cờ đơn giản cộng với cái hàm lương giá 'chất' của quân cờ và 1 số heuristics mà thui !(Em cũng muốn share 1 vài kinh nghiệm về cài thế cờ lúc trung cuộc cho cờ tướng lắm nhưng thui để lúc khác ...).  
- Nói vậy là để các bác thấy việc phân tích lượng giá dựa thế cờ là vô vọng đối với các loại cờ phức tạp khác ... nhưng đối với cờ Caro thì điều này là hoàn toàn đơn giản !!! Thế cờ của Caro chỉ là 2 đường đe doạ giao nhau mà thôi, tức là rơi vào 1 trong 3 cấu hình sau :  
+ Hai đường 4 [1]  
+ Một đường 4 và một đường 3 rỗng 2 đầu [2]  
+ Hai đường 3 rỗng 2 đầu [3]  
và 'Hết', không còn dạng nào nữa vì ví dụ có bác nói đường 4 rỗng 2 đầu thì đó chỉ là bước tiếp theo của [2] hoặc [3] mà thui. Như vậy các bác chỉ cần nhận biết khi nào thì tồn tại các đường trên là được rồi ! Nhưng nhận biết các đường trên xảy ra lúc nào lại là 1 chuyện khác và làm sao để đạt được cấu hình trên nhanh nhất (ít nhất là phải nhanh hơn đối thủ) lại là 1 chuyện khác nữa ... nhưng nói chung, cái nguồn gốc để nhận biết chính là làm sao để nhận biết hoàn toàn trạng thái của 1 ô trên bàn cờ ví dụ như nó có dạng là đường 4 đe doạ hay đường 3 rỗng 2 đầu hay là đường 4 đã bị chận 2 đầu không còn nguy hiểm, hay là đường 4 rỗng 2 đầu (tất thắng), hay là ... Và ... và em thật sự đã tìm ra 1 cách tính toán để chỉ ra tất cả trạng thái đó !!! Các bác hãy chờ xem nhá !!! He he he ...  
  
+Ghi chú các ký hiệu :  
 : Ô trống đang xét  
\* : Ký hiệu chỉ ô trống khác  
O : Quân cờ  
|...| : Cấu hình nhận được trong khoảng này...  
= : Ô 'chết' đối với quân O, tức là đã bị quân X chiếm hoặc là biên của bàn cờ, tức là không đáng quan tâm nữa đối với quân O (ô vô dụng)  
  
+Sau đây là vài trong số tất cả thế cờ mà chương trình sẽ nhận biết được :  
|====OOO=|0 ; Chưa đủ quân số (5 con) mà đã bị 'triệt sản' rùi !  
|==\*OOOOO|0 ; Nhiều quá ... chen chúc nhau 6 con thì làm sao mà ăn !  
|===O\*\*OO|0 ; Không thể nào đánh được đúng 5 con để thắng ...  
|==OO\*\*OO|0 ; ... nên loại ra khỏi quá trình lượng giá !  
|===\*OO==|0 ; Cấu hình 'triệt sản'  
(0) ==> Những cấu hình vô ích, đánh vào chỉ tốn quân !!  
|==\*OOOO\*|1  
|==\*OOOO=|1  
|==O\*OOOO|1 ; Có khả năng thắng nếu kiểm tra thêm ...  
|===\*OOOO|1 ; ... điều kiện là bên phải nhất còn rỗng  
(1) ==> Đây là những cấu hình tất thắng cho quân O  
|\*O\*\*OOO\*|2 ; Gõ vô ô trống ...  
|O\*\*\*OOO\*|2 ; ... đang xét ...  
|=O\*\*OOO\*|2 ; ... là thượng sách !!  
(2) ==> Đây là những cấu hình tất thắng cho quân O nếu đến lượt quân O đi  
|O\*O\*O\*OO|3  
|\*\*\*OO\*O\*|3  
|===OOO\*=|3  
|=\*\*\*OOO=|3  
|=\*\*OO\*O\*|3  
|=OOO\*\*\*\*|3  
(3) ==> Đây là những đường 4 nguy hiểm khi quân O đánh vào ô trống đang xét  
|=\*\*O\*O\*\*|4  
|=\*\*OO\*\*\*|4  
|\*\*O\*O\*\*\*|4  
|=\*\*\*\*OO\*|4  
|==\*\*OO\*=|4  
(4) ==> Đây là những đường 3 rỗng 2 đầu khi quân O đánh vào ô trống đang xét  
  
==> Chiến thắng cho quân O chỉ là sự tìm kiếm 1 tổ hợp chập 2 (tối thiểu) của những đường trên, dĩ nhiên là có xét thêm 1 vài yếu tố như : khi ta ta đánh vào ô này thì sẽ có 2 đường 3 mà đối thủ đã có đường 4 rồi thì có khùng mà uýnh dzô!!  
Thôi thì hẹn các bác ở bài 5 : BƯỚC VÀO KỸ THUẬT LƯỢNG GIÁ vào một ngày đẹp trời gần đây nhất !!

Viết chương trình chơi Caro - Phần 5  
  
**1. Cài đặt CTDL để lưu cấu hình của bàn cờ:**  
\* Ý tưởng :  
- Đầu tiên xét 1 ô trống trên bàn cờ, khi đó, giả sử ta chỉ kiểm tra trên phương nằm ngang (các phương khác tương tự) bao gồm n ô liên tiếp bên trái và n ô liên tiếp bên phải của nó (tức là xét trên 1 hàng ngang có chiều dài 2n+1 ô liên tiếp, khi đó ta có thể thấy được vài điều sau để tóm gọn lại vấn đề :  
+ Ta phải xem xét ô trống này trên 2 phương diện : đối với quân ta hay đối với quân của đối thủ, do đó lúc này thì cách làm cho mỗi bên là tương tự nhau ==> chọn quân ta (quân O).  
+ Việc xét n ô bên trái hay bên phải thì cách làm cũng tương tự nhau (cùng xuất phát từ vị trí ô trống đang xét nhưng chỉ khác nhau về hướng), do đó giả sử chỉ xét một bên (trái).  
+ Với qui ước :  
+ n = 4 và đang kta cho quân O  
+ Dấu X,O : ô đã bị chiếm bởi quân X,O  
+ Dấu \* : ô trống trên bàn cờ  
+ Dấu # : biên của bàn cờ  
+ Dấu ? : ô trống đang xét  
thì cấu hình của n ô đó có thể như sau :  
  
<> Không có ô nào đáng để ta quan tâm nữa :  
Ví dụ : Xét 1 cấu hình trên bàn cờ : \*{/color}\*OOOX{/color}?OOO\*\* : Khi đó ta kiểm tra từ phải sang trái (bắt đầu từ dấu '?'), gặp ngay quân X kế tiếp => do đó lúc này dù 3 ô kế tiếp đều chứa quân O nhưng đối với ô trống đang xét thì nó chẳng có tích sự gì => Tức là bên trái không còn ô nào đáng được quan tâm. Điều này cũng xảy ra khi ô trống đang xét là nằm kế biên hoặc là bên trái của nó đã chứa >= 5 con (đúng 5 con mới được ăn).  
<> Chỉ có 1 ô cần quan tâm.  
Ví dụ : Xét các cấu hình sau : 1) #\*?OO\*\* và 2) \*\*OOX\*?\*\* và 3) OXO?OO\*\* : Khi đó (1) và (2) chỉ còn 1 ô trống kế tiếp bên trái, (3) chỉ còn 1 quân cờ 'O' kế tiếp bên trái => Tức là còn đúng 1 ô cần quan tâm ...  
<> Chỉ còn 2 ô cần quan tâm  
Ví dụ : Xét : 1) OX\*O?\*\*O\* và 2) #\*\*?OO\*\* và 3) OXO\*?O\*\* ...  
<> Có 3 ô cần quan tâm  
Ví dụ : Xét : 1) #OOO?O\*\* và 2) \*XOO\*?\*O\* và 3) OX\*O\*?O\*\* ...  
<> Có 4 ô cần quan tâm  
Ví dụ : Xét : 1) #\*\*\*\*?\*\*\* và 2) XOOOO?\* và 3) #OO\*O?\*\* và 4) XO\*O\*?\*\* ...  
  
==>> Như vậy ô cần quan tâm là những ô trống hoặc là những ô của quân ta mà chưa có sự 'che chắn' (là sao nhỉ :p ) của quân đối thủ ...  
- Và bây giờ, rõ ràng muốn lượng giá 1 thế cờ thì mình phải biết làm sao để đánh giá từng ô trống ngon lành như thế nào, rồi lựa những ô ngon ngon xách ra uýnh ...Tức là làm sao phải lưu lại sư 'quan tâm&đánh giá' này bằng CTDL đàng hoàng, dễ truy cập đến, dễ thay đổi, dễ ... xài mà đừng dễ ... sợ ... hì hì hì ... Beng, beng, beng ! Và sau đây là cách của em :  
+ Xem xét các cấu hình có thể có của 1 bên (trái hay phải) chỉ có thể là :  
  
  
/\*\*\*\* Xét n = 4 \*\*\*\*/  
const char StTable[31][5] = {  
"####", // Hông còn ô nào đáng quan tâm !  
"\*###","O###", // Í! Còn 1 ô nè !!  
"\*\*##","\*O##","O\*##","OO##", // Ố! 2 ô lận !!  
"\*\*\*#","\*\*O#","\*O\*#","\*OO#","O\*\*#","O\*O#","OO\*#"," OOO#", //Á ! 3 ô !  
"\*\*\*\*","\*\*\*O","\*\*O\*","\*\*OO","\*O\*\*","\*O\*O","\*OO\*"," \*OOO", //Chà! Tới 4 ô !  
"O\*\*\*","O\*\*O","O\*O\*","O\*OO","OO\*\*","OO\*O","OOO\*"," OOOO"};  
  
  
\* Phân tích 1 vài các mẫu : Hãy xét từ trái sang phải :  
<> StTable[0] ("####") : Không có ô nào đáng quan tâm vì nó đã bị chiếm bởi quân củaa đối thủ hoặc là ô nằm kế cạnh củaa board.  
<> StTable[1] ("\*###") : Chỉ còn 1 ô trống kế tiếp, còn lại không đáng quan tâm.  
<> StTable[2] ("OO##") : 2 Ô kế tiếp (trên hướng đang xét, ví dụ bên trái) là quân ta, ngoài ra không còn ô nào đáng quan tâm  
<> StTable[30]("OOOO") : Có 4 quân cờ bên ta liên tiếp => Tuyệt vời !  
<> ...  
  
==> Ta định nghĩa theo kiểu này là rất trực quan, thể hiện các cấu hình có thể có với 4 ô liên tiếp trên bàn cờ, mỗi phần tử của 1 mảng StTable là 1 chuỗi có chiều dài không đổi và bằng 4 ... Nhưng mảng này chỉ có ý nghĩa để trình bày trên màn hình mà thôi, nếu không, cách cài đặt theo CTDL này sẽ rất khó để xử lý ==> cải tiến : chỉ lưu con số chỉ vị trí của từng cấu hình mà thôi, ví dụ nếu có số 3 (bắt đầu từ 0) => đó là cấu hình "\*###", có số 30 => "OOOO" ... => cần 1 biến kiểu int thui !... => Đặt tên đại là "Cfg Bên trái/phải" ...  
+ Í!...Mỗi ô có 2 phía, mỗi phía là 1 con số có nghĩa từ 0..30 như trên => Do đó ta có :  
  
  
typedef struct CELL\_PATTERN  
{  
int nBackward; /\* nBackward = 0..30 <== Left side \*/  
int nForward; /\* nForward = 0..30 <== Right side \*/  
}  
PATTERN\_TYPE,\*PPATTERN\_TYPE;  
typedef PATTERN\_TYPE PatternType;  
  
+ Mà quên, mỗi ô lại còn được xét trên 4 phương nữa chứ : (-)ngang(horizontal),(|)dọc(vertical), (\)chéo chính(main diagonal) và (/)chéo phụ(sub diagonal) ==> Phải lưu đến 4 cái như vậy cho mỗi ô ... Và cuối cùng là cho cả 2 bên : quân ta và quân địch => Thêm 2 đứa nữa nhá ==> Và em có (các bác cũng có http://forums.congdongcviet.com/images/smilies/biggrin.gif ...) :  
  
  
PatternType LinePattern[2][BOARD\_SIZE\*4];// member of CBoard class  
  
  
- Như vậy xét trên 1 phương thì ta sẽ có tổ hợp 31x31 cấu hình có thể có cho 1 ô !  
Ví dụ :  
+ Cặp (0,29) biểu thị cho cấu hình 9 ô liên tiếp như sau : "####OOO\*";  
+ Cặp (5,27)= "##\*OOO\*\*";  
+ (0,30) = "####OOOO";  
+ (1,13) = "###\*OO\*#";  
+ (10,26) = "#OO\*O\*OO";  
+ (12,29) = "#O\*OOOO\*";  
... trong đó ô vuông () biểu thị cho ô trống đang xét (tức là y chang dấu '?' đã trình bày ở trên) ...  
  
\*Code : Một đoạn chương trình nhỏ để quan sát tất cả các cấu hình này :  
  
  
// Xuất ra file : Out.txt tất cả cấu hình cần xem :  
#include <fstream.h>  
const char StTable[31][5] = {  
"####",  
"\*###","O###",  
"\*\*##","\*O##","O\*##","OO##",  
"\*\*\*#","\*\*O#","\*O\*#","\*OO#","O\*\*#","O\*O#","OO\*#"," OOO#",  
"\*\*\*\*","\*\*\*O","\*\*O\*","\*\*OO","\*O\*\*","\*O\*O","\*OO\*"," \*OOO",  
"O\*\*\*","O\*\*O","O\*O\*","O\*OO","OO\*\*","OO\*O","OOO\*"," OOOO"};  
void main()  
{  
ofstream out\_all("Out.txt");  
for (int i = 0;i < 31;i++)  
{  
for (int j = 0;j < 31;j++)  
{  
out\_all << "(" << i << "," << j << ")" << "\t: |";  
for (int k = 3;k >= 0;k--)  
out\_all << StTable[i][k] << flush;  
out\_all << char(0x19) << flush;  
out\_all << StTable[j] << "|" << endl;  
}  
}  
} // end  
  
  
Ghi chú : Hổng phải em viết cái đoạn này ra là để xem chơi mà thôi đâu ! Sẽ có việc làm với nó sau này : đánh giá thế nào là "hảo hảo ..." một cách trực quan dựa trên sự chọn lựa của mình ..., vì lý do này, có thể xem cấu hình (a,b) và (b,a) là 1 (chỉ thay đổi vị trí thôi) để sau này dễ làm hơn ...  
Ví dụ nó in cái rẹt ra tương tự như vậy :  
(5,29) : |##\*OOOO\*|1  
(11,27) : |#\*\*OOO\*\*|2  
(19,27) : |\*\*O\*OO\*\*|3  
(11,19) : |#\*\*O\*O\*\*|4  
(7,23) : |#\*\*\*O\*\*\*|5  
=> Các số 1,2,3,4,5 là em thêm vào để đánh giá sự tốt xấu của thế cờ kế tiếp nếu quân O đánh vào vị trí ô vuông đang xét ... => Từ đó viết 1 chương trình để đọc & xử lý lại file này thì ta sẽ có 1 cơ sở dữ liệu để lưu được sự tốt xấu của từng thế cờ ...

2. Làm việc với CTDL này:  
- Việc truy xuất đến thì chắc dễ rồi, vì chỉ cần biết nForward Và nBackward là biết ngay nó đang có dạng (configuration) như thế nào, sau đó tuỳ theo dạng này mà lượng giá ... Do đó mục tiêu khó khăn đặt ra là phải cập nhật các cấu hình của các ô sau mỗi nước đi của cả 2 bên sao cho cấu hình này nó phản ánh đúng & chính xác cấu hình thật sự của từng ô trống trên bàn cờ...việc làm này cũng phải nhanh vì nó sẽ nằm trong phần gọi đệ qui. Em đưa ra cách cài đặt như sau :  
+ Khai báo 1 mảng để lưu các biến đổi từ cấu hình này sang cấu hình khác sau khi một ô trống bị thay đổi trạng thái (tức là thành quân ta hay quân đối thủ), mảng này sẽ có nhiệm vụ chuyển từ 1 cấu hình có sẵn sang 1 cấu hình mới (dĩ nhiên nó vẫn sẽ nằm trong tập 0..30) sau khi đánh 1 quân cờ ...  
+ Bây giờ giả sử ta xét trên 1 hàng ngang 9 ô, ô thứ 5 (ở chính giữa) là ô trống đang xét), 4 ô ở bên trái (từ 1 đến 4) và 4 ô ở bên phải (từ 6 đến 9) tương ứng là 1 cặp cấu hình (trong số tổ hợp 31 cấu hình ở trên) sẽ có dạng sau.  
<> Vị trí đang xét là ô trống thứ 5 (hình vuông)  
Vị trí : 1 2 3 4 5 6 7 8 9  
Mảng a[i] : x x x x x x x x  
Trong đó x là 1 trạng thái chưa biết(nằm trong tập : {O,X,#,\*}) nhưng không quan trọng vì ta chỉ quan tâm các cấu hình dưới dạng các con số mà thôi! (Rõ ràng là vậy vì 4 chữ x bên trái ô vuông là thành phần nBackward, 4 chữ x bên phải là nForward, cả nBackward & nForward đều là 1 con số thuộc 0..30 nằm trong cấu trúc PATTERN đã nói ở trên).  
==> Đánh 1 quân cờ vào vị trí ô trống đang xét ==> Cập nhật lại cấu hình bàn cờ :  
<> Cập nhật cho 4 ô ở phía bên trái của ô đang xét bằng cách chạy 1 vòng lặp 4 lần, bắt đầu từ vị trí số 4 (kế bên trái ô vuông), giảm dần, nếu gặp 1 ô trống thì ta sẽ cập nhật lại mới cấu hình của phía bên phải nó (tức là cập nhật lại thành phần nForward của ô trống đó), nếu gặp quân ta thì chạy tiếp (hông làm gì cả), nếu gặp quân đối thủ hoặc bắt đầu đụng biên thì break.  
<> Cập nhật lại 4 ô bên phải của ô đang xét thì tương tự và ngược lại ...  
==> Bây giờ câu hỏi đặt ra là tại sao chỉ là vòng lặp 1->4 (thật ra là 0->3) là đủ cập nhật lại toàn bộ cấu hình của bàn cờ sau nước đi vào ô trống hình vuông số 5 ??? Thật ra là việc thay đổi ô trống số 5 thì chỉ làm thay đổi đúng 8 ô trống trên hàng ngang mà kế cận nó, bao gồm 4 ô bên trái và 4 ô bên phải, vì rõ ràng là cấu hình bên trái (nBackward) của ô số 9 thì có thể phụ thuộc vào giá trị ô số 5 nhưng những ô >= 10 thì không còn phụ thuộc nữa (nhớ lại là em đang xét với n = 4 thôi nhá!)  
+ Việc cập nhật lại 1 cấu hình thật sự là rất đơn giản như sau :  
<> Lập sẵn 1 bảng UpdateFriendPattern (cập nhật mẫu cho quân ta).  
<> Lập sẵn 1 bảng UpdateEnemyPattern (cập nhật mẫu cho quan đối thủ).  
<> Tra bảng để chuyển cấu hình tương ứng với ô đang xét.  
+ Thiết kế bảng cập nhật mẫu như sau : (Có thể khi em post lên cái bảng này sẽ lộn xộn rất khó đọc, các bác chịu khó copy nó ra file \*.txt thì mới dễ thấy ... còn nếu không nhờ mấy bác mod sửa dùm...)  
  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* UPDATE FRIEND PATTERN \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
N Form-1 Form-2 Form-3 Form-4 Form-5 Lookup-Table  
----------------------------------------------------------------------------------------------  
0 "####" const const const const {0,0,0,0}  
1 "\*###" "O###" const const const {2,1,1,1}  
2 "O###" const const const const {2,2,2,2}  
3 "\*\*##" "O\*##" "\*O##" const const {5,4,3,3}  
4 "\*O##" "OO##" const const const {6,4,4,4}  
5 "O\*##" const "OO##" const const {0,6,5,5}  
6 "OO##" const const const const {6,6,6,6}  
7 "\*\*\*#" "O\*\*#" "\*O\*#" "\*\*O#" const {11,9,8,7}  
8 "\*\*O#" "O\*O#" "\*OO#" const const {12,10,8,8}  
9 "\*O\*#" "OO\*#" const "\*OO#" const {13,9,10,9}  
10 "\*OO#" "OOO#" const const const {14,10,10,10}  
11 "O\*\*#" const "OO\*#" "O\*O#" const {11,13,12,11}  
12 "O\*O#" const "OOO#" const const {12,14,12,12}  
13 "OO\*#" const const "OOO#" const {13,13,14,13}  
14 "OOO#" const const const const {14,14,14,14}  
15 "\*\*\*\*" "O\*\*\*" "\*O\*\*" "\*\*O\*" "\*\*\*O" {23,19,17,16}  
16 "\*\*\*O" "O\*\*O" "\*O\*O" "\*\*OO" const {24,20,18,16}  
17 "\*\*O\*" "O\*O\*" "\*OO\*" const "\*\*OO" {25,21,17,18}  
18 "\*\*OO" "O\*OO" "\*OOO" const const {26,22,18,18}  
19 "\*O\*\*" "OO\*\*" const "\*OO\*" "\*O\*O" {27,19,21,20}  
20 "\*O\*O" "OO\*O" const "\*OOO" const {28,20,22,20}  
21 "\*OO\*" "OOO\*" const const "\*OOO" {29,21,21,22}  
22 "\*OOO" "OOOO" const const const {30,22,22,22}  
23 "O\*\*\*" const "OO\*\*" "O\*O\*" "O\*\*O" {23,27,25,24}  
24 "O\*\*O" const "OO\*O" "O\*OO" const {24,28,26,24}  
25 "O\*O\*" const "OOO\*" const "O\*OO" {25,29,25,26}  
26 "O\*OO" const "OOOO" const const {26,30,26,26}  
27 "OO\*\*" const const "OOO\*" "OO\*O" {27,27,29,28}  
28 "OO\*O" const const "OOOO" const {28,28,30,28}  
29 "OOO\*" const const const "OOOO" {29,29,29,30}  
30 "OOOO" const const const const {30,30,30,30}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* UPDATE ENEMY PATTERN \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
N Form-1 Form-2 Form-3 Form-4 Form-5 Lookup-Table  
-----------------------------------------------------------------------------------------------  
0 "####" const const const const {0,0,0,0}  
1 "\*###" "####" const const const {0,1,1,1}  
2 "O###" const const const const {2,2,2,2}  
3 "\*\*##" "####" "\*###" const const {0,1,3,3}  
4 "\*O##" "####" const const const {0,4,4,4}  
5 "O\*##" const "O###" const const {5,2,5,5}  
6 "OO##" const const const const {6,6,6,6}  
7 "\*\*\*#" "####" "\*###" "\*\*##" const {0,1,3,7}  
8 "\*\*O#" "####" "\*###" const const {0,1,8,8}  
9 "\*O\*#" "####" const "\*O##" const {0,9,4,9}  
10 "\*OO#" "####" const const const {0,10,10,10}  
11 "O\*\*#" const "O###" "O\*##" const {11,2,5,11}  
12 "O\*O#" const "O###" const const {12,2,12,12}  
13 "OO\*#" const const "OO##" const {13,13,6,13}  
14 "OOO#" const const const const {14,14,14,14}  
15 "\*\*\*\*" "####" "\*###" "\*\*##" "\*\*\*#" {0,1,3,7}  
16 "\*\*\*O" "####" "\*###" "\*\*##" const {0,1,3,16}  
17 "\*\*O\*" "####" "\*###" const "\*\*O#" {0,1,17,8}  
18 "\*\*OO" "####" "\*###" const const {0,1,18,18}  
19 "\*O\*\*" "####" const "\*O##" "\*O\*#" {0,19,4,9}  
20 "\*O\*O" "####" const "\*O##" const {0,20,4,20}  
21 "\*OO\*" "####" const const "\*OO#" {0,21,21,10}  
22 "\*OOO" "####" const const const {0,22,22,22}  
23 "O\*\*\*" const "O###" "O\*##" "O\*\*#" {23,2,5,11}  
24 "O\*\*O" const "O###" "O\*##" const {24,2,5,24}  
25 "O\*O\*" const "O###" const "O\*O#" {25,2,25,12}  
26 "O\*OO" const "O###" const const {26,2,26,26}  
27 "OO\*\*" const const "OO##" "OO\*#" {27,27,6,13}  
28 "OO\*O" const const "OO##" const {28,28,6,28}  
29 "OOO\*" const const const "OOO#" {29,29,29,14}  
30 "OOOO" const const const const {30,30,30,30}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/