

# 人工智能课程大作业二

## $\alpha$ - $\beta$ 剪枝的四子棋策略

沈磊

物理系 2015 级

### 一、四子棋背景介绍

在人工智能领域的研究中，博弈问题一直是一个代表性问题，研究历史也是非常久远。因为在通常大家的认识当中，博弈问题是比较能够考验一个人的智力水平的。如果计算机和人进行博弈，是一种典型的人工智能的体现。而且随着博弈问题的不断研究，研究出了很多人工智能方面的算法，也极大地推动了机器智能的进步。

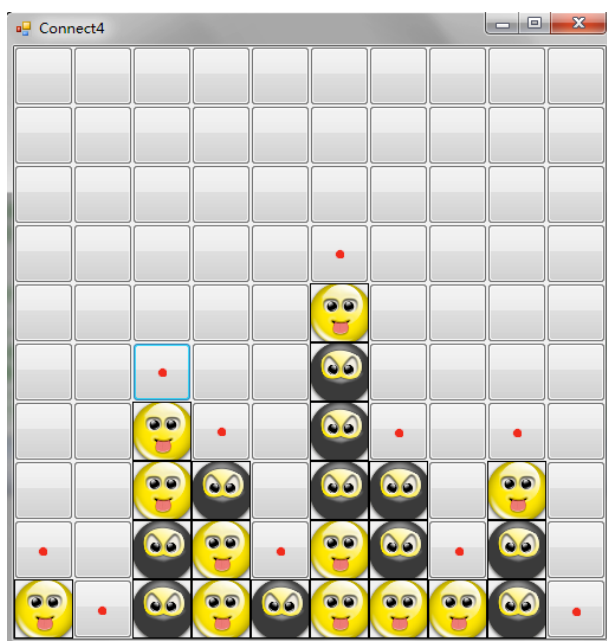
棋类问题是博弈问题的一个典型例子。在棋类博弈中，竞技双方的条件相对公平，对方的情况是透明的，双方的局势评价标准是一样的。这类问题需要用到的就是各种搜索算法，简而言之，谁对场上局势把握更准确，谁能想的比对方更远，谁就更有优势。所以在本次大作业中主要用到的就是**极小极大算法**和 **$\alpha$ - $\beta$ 剪枝算法**。

棋类问题有很多，围棋最复杂，中国象棋、国际象棋、五子棋、四子棋复杂程度依次递减。棋类博弈过程可以看成是一个庞大的博弈树，刚刚说的“复杂”指的是搜索的博弈树每展开一步都的多少，子节点多就是“复杂”，子节点少就是“简单”。所以围棋每拓展依次会有很多节点，而四子棋拓展依次节点不会那么多（当然这和棋盘大小有关系）。所以本次作业以四子棋作为我学习人工智能搜索算法的一个检验。

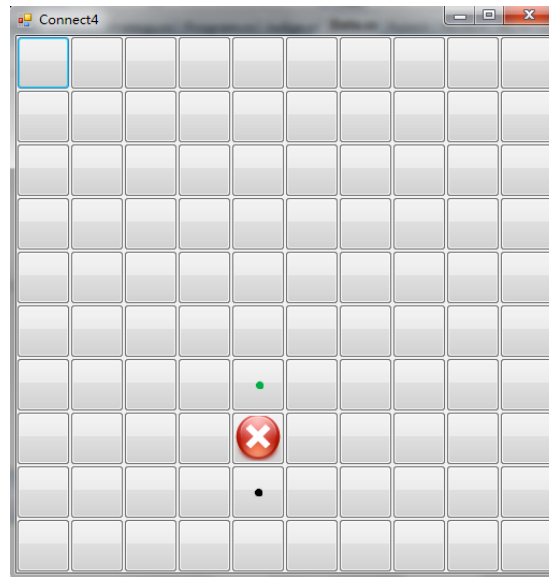
### 二、四子棋规则及算法分析

#### 1、规则介绍

本次作业的四子棋规则不同于普通四子棋的规则。如图：



首先棋局是不定大小的长宽在[9,12]之间。每一步棋只能走每一列的顶部可走位置，如果满了，不可走。胜负判定还是水平、竖直和两个斜方向，任意成四连子即获胜。除此之外，还有一个不可落子点的约束，如图：



其他要求都不变，只在到达该不可落子点之后就跳过，下一步可走在该不可落子点的上面，也即图中的黑色和绿色点是先后可走的点。

普通的四子棋，如果棋盘比较小，根据先后手有必胜策略。但是在本次作业中，由于不可落子点的不确定约束，必胜策略不是很容易应用。而且落子有方向要求（会在评价函数设计部分描述），所以和常规四子棋也不一样。

## 2、算法分析

此类型博弈问题，经常用的算法是极小极大搜索， $\alpha$ - $\beta$ 剪枝算法，蒙特卡洛树搜索算法。本次作业采用极小极大搜索和 $\alpha$ - $\beta$ 剪枝算法。

### (1)、极小极大搜索算法

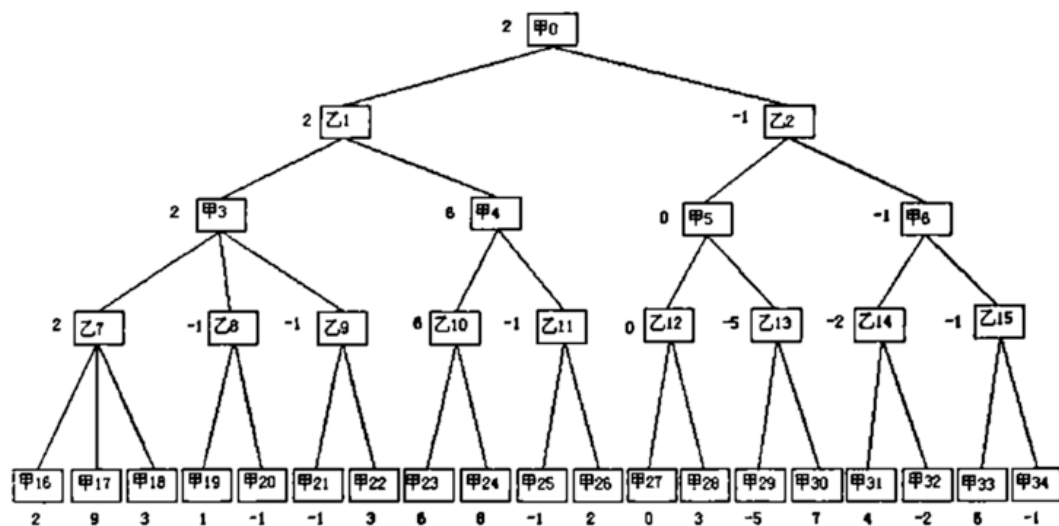
假设我们先让两个人进行四子棋博弈，过程如图：



我方和敌方轮流走子。我在走子的时候会对场上形势做一个判断：先找有没有成四连子的走法，如果有，那我就赢了。如果没有，再找有没有有利于我方的走法，如果有，再看看该步会不会让对方“更容易”赢，如果不会有利于对方，那可以走改点。如果有利于对方，看看别的点能不能，同样按照此判断。

注意上述走子思考方式有几个关键点：1、双方都清楚场上形势，都可以对场上形势做一个评价，而且假设双方同等厉害，双方对局势的评价结果应该是一样的（可用相同的评价函数）。2、同样的棋局，对我方有利，那么就对敌方不利。所以在走子的时候我方就要走评价函数的极大点，敌方走评价函数的极小点。3、我方在走子的时候要往后考虑几步，这样才能走出更加有利于我方的走法。

所以就引出了我们引用的基本算法：极小极大搜索算法。如图：



正如上图所示，假设评价函数越大就是对甲方越有利，评价函数越小就是对乙方越有利。所以可以称甲的层为 max 层，乙的层为 min 层。在博弈过程中，甲要使得评价函数越大越好，所以从甲层到乙层是搜索极大值，而反过来，乙层到甲层是搜索极小值。还有一点非常重要，下棋双方往往对后来子节点的评分会更加准确。因为越往后，双方的输赢就越明显。所以每

个节点的分值应当尽量由子节点来决定，此即所谓评价函数的反向传播方法，该方法也是 $\alpha$ - $\beta$ 剪枝的基础。当考虑完若干层之后，甲就选他的子节点评价函数值最大的，即最有利于甲的；反过来，乙就选择评价函数最小的节点。上述过程就是极小极大搜索算法过程。

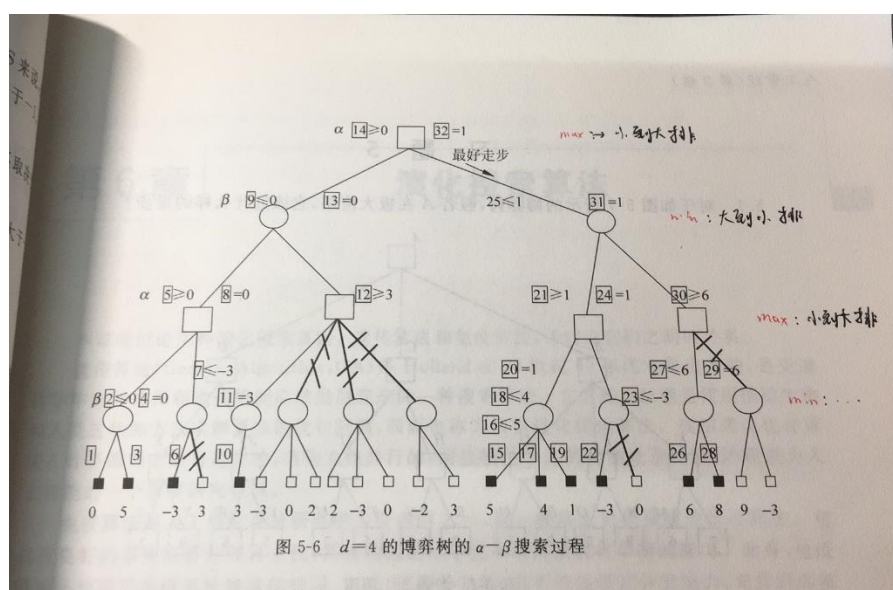
## (2)、 $\alpha$ - $\beta$ 剪枝算法

上述搜索算法很好，假设评价函数得当，计算机的计算足够强大，考虑的层数足够多，那么棋力可以预期会非常强。普通玩家大概可以到4层，比较好的选手可以考虑6层，更好的选手可以考虑到8层（当然上述说的是在棋局评价函数相当的时候）。

为什么大多数人考虑最多也不到10层呢？因为，这个博弈搜索树是指数级增长的，之后每增长一级，数量会是之前的倍数（围棋每一层状态数那么多，可想而知有多难）。所以不能这样一直加深考虑的层数。

考虑一种情况，当我走一步棋的时候有很明显对我方有利的走法（比如制胜走法），那么我就只在这一支上面拓展就行了。同理，如果这一步明显更有利于对方，那么我就不用在这一支上拓展了。这样可以减少计算量，而且和全部拓展效果是一样的。这种可以不考虑某些分支的算法就是著名的 $\alpha$ - $\beta$ 剪枝算法。

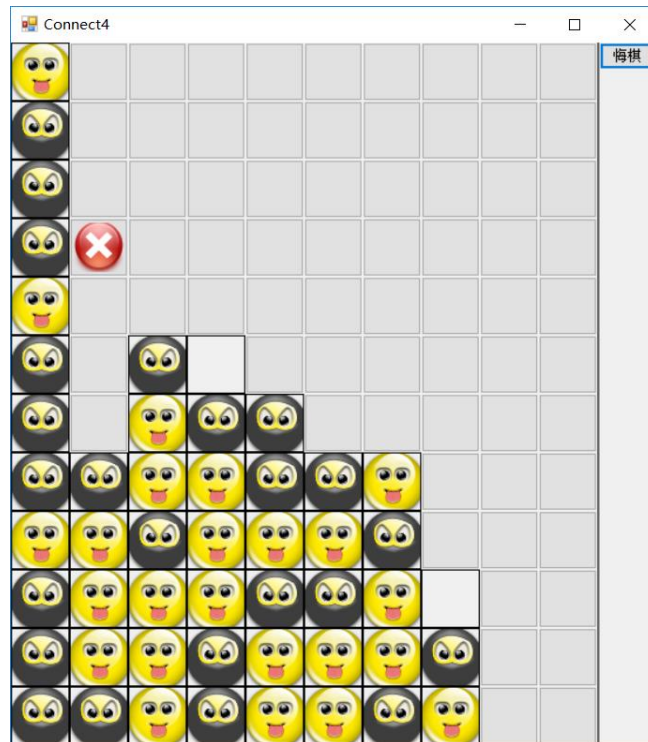
用算法的语言描述就是：我们把max层的极小值叫做 $\alpha$ ，把min层极大值叫做 $\beta$ 。当max层的极小值 $\alpha$ 大于其子节点min层的极大值 $\beta$ ，那么该分支就可以不用再接着拓展了，这叫 $\alpha$ 剪枝；当min层的极大值 $\beta$ 小于其子节点man层的极小值 $\alpha$ ，那么该分支就可以不用再接着拓展了，这叫 $\beta$ 剪枝。过程如图所示：



大体原则就是，在能把函数值传递给父节点就尽早传递，这样就可以跟其他的子节点进行剪枝判断。

## (3)、本次作业算法设计

本次作业采用的就是极小极大搜索算法基础上的 $\alpha$ - $\beta$ 剪枝算法。该方法中有几个关键点很重要：1、评价函数要尽量好，好的评价函数意味着对局势把握好。2、层数尽可能多。3、剪枝效果尽可能好。



#### (a)、评价函数：

通过我不断和已有策略进行下棋，总结出对于棋盘局势的评级函数。具体评价包含以下一个子的形状：四连子，活三，活二，活一，死三，死二，死一。“活”即为最大可走的窗口大于3。“死”就是最大窗口小于等于3。每个分数成倍递减（具体可看代码）。

很重要的一点，棋的走法是数值向下的，横竖和两个斜线方向的四个方向不等价的。竖直方向的活子永远都是“半活”（最下面不能走子），所以评价函数在数值方向相对其他方向比较小。

很重要一点，评价函数永远一般都不会是尽善尽美的。这时候有一种局势会非常重要，也就是“杀棋”。只要把局盘中的杀棋给出，其实其他状态的评分不是非常接近完美，也可以用加深层数的办法来增加杀棋出现的概率，这样就可以把其他无关痛痒的局势的评分影响降到最低，这样会更加接近真实的评价值。

#### (b) 负极大值搜索：

细想一下，甲方和乙方的评价函数都是一样的，把甲子和乙子的评分相减，正的就是甲子，负的评分，就是乙子的评分。这样就可以运用一种从 $\alpha$ - $\beta$ 剪枝算法改进的一种负极大搜索算法。因为两种极值搜索方式的算法形式是一样的，这样就可以把极大值搜索和极小值搜索写成一个函数，每次变换角色的时候，只要把甲的参数交换传给乙来调用负极大搜索函数就可以了（可看后面实现部分）。

### 三、代码实现和算法改进

#### 1、实现过程

##### (1)、设置全局变量用于更新保存每次拓展的情况

对于 board, top, lastX, lastY 都是在拓展和评价函数计算的时候要用的，设成全局变量。

```

1  #include <iostream>
2  #include <conio.h>
3  #include <atlstr.h>
4  #include <ctime>
5  #include <cmath>
6  #include <cstdlib>
7  #include "Point.h"
8  #include "Strategy.h"
9  #include "Judge.h"
10
11  using namespace std;
12
13  static int lastx;
14  static int lasty;
15  static int next_step[2]; //存储下一步的走步
16  static int *mytop; //存储每一次拓展后的top
17  static int **myboard; //存储每次拓展后的board
18  static int DEPTH = 8; //设置剪枝深度
19
20
21  int evaluate_board(bool type, const int M, const int N); //局面评价函数
22  bool hasThreatPoint(bool type, int x, int y, const int M, const int N); //判断一个点是否是威胁点
23  bool userWin(const int x, const int y, const int M, const int N, int* const* board); //用户胜
24  bool machineWin(const int x, const int y, const int M, const int N, int* const* board); //策略胜
25  bool isTie(const int N, const int* top); //平局
26
27
28  int negmaxsearch(bool mytype, int mydepth, int myalpha, int mybeta, const int M, const int N) {...}
111

```

Strategy (全局范围)

```

151  /*
152  //Add your own code below
153  /* ... */
154  //我的代码:
155  board[noX][noY] = -1; //把不可落子点与无落子点区别出来
156  //初始化myboard
157  myboard = new int*[M];
158  for (int i = 0; i < M; i++) {
159      myboard[i] = new int[N];
160      for (int j = 0; j < N; j++) {
161          myboard[i][j] = board[i][j];
162      }
163  }
164  //_cprintf("%d\t ", myboard[noX][noY]);
165  //初始化mytop
166  mytop = new int[N];
167  for (int i = 0; i < N; i++) {
168      mytop[i] = top[i];
169  }
170  //初始化lastx, lasty
171  lastx = lastX;
172  lasty = lastY;
173  //_cprintf("%d\t%d\t ", lastx, lasty);
174  //处理必杀棋和威胁棋
175  int xindex, yindex;
176  bool isabcut = true; //标记, 用于判断用不用ab剪枝
177  for (int i = 0; i < N; i++) {
178      if (mytop[i] == 0) continue;
179      else {
180          xindex = mytop[i] - 1;
181      }
182  }
183  }
184  }
185  }
186  }
187  }
188  }
189  }
190  }
191  }
192  }
193  }
194  }
195  }
196  }
197  }
198  }
199  }
200  }
201  }
202  }
203  }
204  }
205  }
206  }
207  }
208  }
209  }
210  }
211  }
212  }
213  }
214  }
215  }
216  }
217  }
218  }
219  }
220  }
221  }
222  }
223  }
224  }
225  }
226  }
227  }
228  }
229  }
230  }
231  }
232  }
233  }
234  }
235  }
236  }
237  }
238  }
239  }
240  }
241  }
242  }
243  }
244  }
245  }
246  }
247  }
248  }
249  }
250  }
251  }
252  }
253  }
254  }
255  }
256  }
257  }
258  }
259  }
260  }
261  }
262  }
263  }
264  }
265  }
266  }
267  }
268  }
269  }
270  }
271  }
272  }
273  }
274  }
275  }
276  }
277  }
278  }
279  }
280  }
281  }
282  }
283  }
284  }
285  }
286  }
287  }
288  }
289  }
290  }
291  }
292  }
293  }
294  }
295  }
296  }
297  }
298  }
299  }
300  }
301  }
302  }
303  }
304  }
305  }
306  }
307  }
308  }
309  }
310  }
311  }
312  }
313  }
314  }
315  }
316  }
317  }
318  }
319  }
320  }
321  }
322  }
323  }
324  }
325  }
326  }
327  }
328  }
329  }
330  }
331  }
332  }
333  }
334  }
335  }
336  }
337  }
338  }
339  }
340  }
341  }
342  }
343  }
344  }
345  }
346  }
347  }
348  }
349  }
350  }
351  }
352  }
353  }
354  }
355  }
356  }
357  }
358  }
359  }
360  }
361  }
362  }
363  }
364  }
365  }
366  }
367  }
368  }
369  }
370  }
371  }
372  }
373  }
374  }
375  }
376  }
377  }
378  }
379  }
380  }
381  }
382  }
383  }
384  }
385  }
386  }
387  }
388  }
389  }
390  }
391  }
392  }
393  }
394  }
395  }
396  }
397  }
398  }
399  }
400  }
401  }
402  }
403  }
404  }
405  }
406  }
407  }
408  }
409  }
410  }
411  }
412  }
413  }
414  }
415  }
416  }
417  }
418  }
419  }
420  }
421  }
422  }
423  }
424  }
425  }
426  }
427  }
428  }
429  }
430  }
431  }
432  }
433  }
434  }
435  }
436  }
437  }
438  }
439  }
440  }
441  }
442  }
443  }
444  }
445  }
446  }
447  }
448  }
449  }
450  }
451  }
452  }
453  }
454  }
455  }
456  }
457  }
458  }
459  }
460  }
461  }
462  }
463  }
464  }
465  }
466  }
467  }
468  }
469  }
470  }
471  }
472  }
473  }
474  }
475  }
476  }
477  }
478  }
479  }
480  }
481  }
482  }
483  }
484  }
485  }
486  }
487  }
488  }
489  }
490  }
491  }
492  }
493  }
494  }
495  }
496  }
497  }
498  }
499  }
500  }
501  }
502  }
503  }
504  }
505  }
506  }
507  }
508  }
509  }
510  }
511  }
512  }
513  }
514  }
515  }
516  }
517  }
518  }
519  }
520  }
521  }
522  }
523  }
524  }
525  }
526  }
527  }
528  }
529  }
530  }
531  }
532  }
533  }
534  }
535  }
536  }
537  }
538  }
539  }
540  }
541  }
542  }
543  }
544  }
545  }
546  }
547  }
548  }
549  }
550  }
551  }
552  }
553  }
554  }
555  }
556  }
557  }
558  }
559  }
560  }
561  }
562  }
563  }
564  }
565  }
566  }
567  }
568  }
569  }
570  }
571  }
572  }
573  }
574  }
575  }
576  }
577  }
578  }
579  }
580  }
581  }
582  }
583  }
584  }
585  }
586  }
587  }
588  }
589  }
590  }
591  }
592  }
593  }
594  }
595  }
596  }
597  }
598  }
599  }
600  }
601  }
602  }
603  }
604  }
605  }
606  }
607  }
608  }
609  }
610  }
611  }
612  }
613  }
614  }
615  }
616  }
617  }
618  }
619  }
620  }
621  }
622  }
623  }
624  }
625  }
626  }
627  }
628  }
629  }
630  }
631  }
632  }
633  }
634  }
635  }
636  }
637  }
638  }
639  }
640  }
641  }
642  }
643  }
644  }
645  }
646  }
647  }
648  }
649  }
650  }
651  }
652  }
653  }
654  }
655  }
656  }
657  }
658  }
659  }
660  }
661  }
662  }
663  }
664  }
665  }
666  }
667  }
668  }
669  }
670  }
671  }
672  }
673  }
674  }
675  }
676  }
677  }
678  }
679  }
680  }
681  }
682  }
683  }
684  }
685  }
686  }
687  }
688  }
689  }
690  }
691  }
692  }
693  }
694  }
695  }
696  }
697  }
698  }
699  }
700  }
701  }
702  }
703  }
704  }
705  }
706  }
707  }
708  }
709  }
710  }
711  }
712  }
713  }
714  }
715  }
716  }
717  }
718  }
719  }
720  }
721  }
722  }
723  }
724  }
725  }
726  }
727  }
728  }
729  }
730  }
731  }
732  }
733  }
734  }
735  }
736  }
737  }
738  }
739  }
740  }
741  }
742  }
743  }
744  }
745  }
746  }
747  }
748  }
749  }
750  }
751  }
752  }
753  }
754  }
755  }
756  }
757  }
758  }
759  }
760  }
761  }
762  }
763  }
764  }
765  }
766  }
767  }
768  }
769  }
770  }
771  }
772  }
773  }
774  }
775  }
776  }
777  }
778  }
779  }
780  }
781  }
782  }
783  }
784  }
785  }
786  }
787  }
788  }
789  }
790  }
791  }
792  }
793  }
794  }
795  }
796  }
797  }
798  }
799  }
800  }
801  }
802  }
803  }
804  }
805  }
806  }
807  }
808  }
809  }
810  }
811  }
812  }
813  }
814  }
815  }
816  }
817  }
818  }
819  }
820  }
821  }
822  }
823  }
824  }
825  }
826  }
827  }
828  }
829  }
830  }
831  }
832  }
833  }
834  }
835  }
836  }
837  }
838  }
839  }
840  }
841  }
842  }
843  }
844  }
845  }
846  }
847  }
848  }
849  }
850  }
851  }
852  }
853  }
854  }
855  }
856  }
857  }
858  }
859  }
860  }
861  }
862  }
863  }
864  }
865  }
866  }
867  }
868  }
869  }
870  }
871  }
872  }
873  }
874  }
875  }
876  }
877  }
878  }
879  }
880  }
881  }
882  }
883  }
884  }
885  }
886  }
887  }
888  }
889  }
890  }
891  }
892  }
893  }
894  }
895  }
896  }
897  }
898  }
899  }
900  }
901  }
902  }
903  }
904  }
905  }
906  }
907  }
908  }
909  }
910  }
911  }
912  }
913  }
914  }
915  }
916  }
917  }
918  }
919  }
920  }
921  }
922  }
923  }
924  }
925  }
926  }
927  }
928  }
929  }
930  }
931  }
932  }
933  }
934  }
935  }
936  }
937  }
938  }
939  }
940  }
941  }
942  }
943  }
944  }
945  }
946  }
947  }
948  }
949  }
950  }
951  }
952  }
953  }
954  }
955  }
956  }
957  }
958  }
959  }
960  }
961  }
962  }
963  }
964  }
965  }
966  }
967  }
968  }
969  }
970  }
971  }
972  }
973  }
974  }
975  }
976  }
977  }
978  }
979  }
980  }
981  }
982  }
983  }
984  }
985  }
986  }
987  }
988  }
989  }
990  }
991  }
992  }
993  }
994  }
995  }
996  }
997  }
998  }
999  }
1000  }

```

## (2)、策略方制胜位置和地方威胁位置判断

如果有制胜位置，立刻走；然后如果有威胁位置，立刻堵上。



```

182     lasty = lastY;
183     // _cprintf("%d\t%d\t", lastx, lasty);
184     // 处理必杀棋和威胁棋
185     int xindex, yindex;
186     bool isabcut = true; // 标记, 用于判断用不用ab剪枝
187     for (int i = 0; i < N; i++) {
188         if (mytop[i] == 0) continue;
189         else {
190             xindex = mytop[i] - 1;
191             yindex = i;
192             if (hasThreatPoint(true, xindex, yindex, M, N)) { // 判断必杀棋
193                 next_step[0] = mytop[i] - 1;
194                 next_step[1] = i;
195                 isabcut = false;
196             }
197             else {
198                 if (hasThreatPoint(false, xindex, yindex, M, N)) { // 判断威胁棋
199                     next_step[0] = mytop[i] - 1;
200                     next_step[1] = i;
201                     isabcut = false;
202                 }
203             }
204         }
205     }
206
207     // 在策略先手时, 把第一颗子放中间远离不可落子点的位置
208     if (!(lastX >= 0 && lastX < M && lastY >= 0 && lastY < N)) {

```

(3)、如果策略方先手, 走远离 noY 位置, 并且尽量靠近棋盘中心

```

207     // 在策略先手时, 把第一颗子放中间远离不可落子点的位置
208     if (!(lastX >= 0 && lastX < M && lastY >= 0 && lastY < N)) {
209         if ((N - 1 - noY) > noY) {
210             next_step[0] = mytop[N-4] - 1;
211             next_step[1] = N-4;
212             isabcut = false;
213         }
214         else {
215             next_step[0] = mytop[3] - 1;
216             next_step[1] = 3;
217             isabcut = false;
218         }
219         // _cprintf("lastX = %d\nlastY = %d\n", lastX, lastY);
220     }
221     if (isabcut) {
222         int score = negmaxsearch(true, DEPTH, -999999999, 999999999, M, N);
223     }
224     // _cprintf("%d\t", score);
225     // 把得到的下一步走法传给x,y
226     x = next_step[0];
227     y = next_step[1];

```

(4)、如果上述特殊情况都没有, 那么就是蛮长的 ab 剪枝的过程了

```

220     }
221     if (isabcut) {
222         int score = negmaxsearch(true, DEPTH, -999999999, 999999999, M, N);
223     }
224     // _cprintf("%d\t", score);
225     // 把得到的下一步走法传给x,y
226     x = next_step[0];
227     y = next_step[1];
228     delete[] mytop;
229     for (int i = 0; i < M; i++) {
230         delete[] myboard[i];
231     }
232     delete[] myboard;
233
234     //////////////////////////////////////
235     /*
236     不要更改这段代码
237     */
238     clearArray(M, N, board);
239     return new Point(x, y);
240 }
241
242

```

(5)、下图就是 ab 剪枝的负极大算法的实现

通过递归拓展分支, 当达到局面确定或是达到拓展深度, 再一层一层递归回原始节点。

```

28 int negmaxsearch(bool mytype, int mydepth, int myalpha, int mybeta, const int M, const int N) {
29     int score = 0;
30     //判断局面:
31     if (machineWin(lastx, lasty, M, N, myboard) || userWin(lastx, lasty, M, N, myboard) || mydepth == 0 || isTie(0, mytop)) {
32         return evaluate_board(mytype, M, N);
33     }
34     //剪枝排序! 越靠近lasty, 优先拓展, 可以极大提升效率。深度可再加深2层!
35     int* sortindex = new int[N];
36     int count = 0;
37     for (int delta = 0; delta < N; delta++) { ... }
38
39     //节点拓展
40     for (int i = 0; i < N; i++) {
41         if (mytop[sortindex[i]] == 0) continue; //满的列不拓展
42         //备份拓展前的点
43         int backx = lastx;
44         int backy = lasty;
45         int* backtop = new int[N];
46         for (int j = 0; j < N; j++) {
47             backtop[j] = mytop[j];
48         }
49         int** backboard = new int*[M];
50         for (int p = 0; p < M; p++) {
51             backboard[p] = new int[N];
52             for (int q = 0; q < N; q++) {
53                 backboard[p][q] = myboard[p][q];
54             }
55         }
56         //更新mytop和myboard, lastx, lasty
57         lastx = mytop[sortindex[i]] - 1;
58         lasty = sortindex[i];
59         if (mytype) myboard[mytop[sortindex[i]] - 1][sortindex[i]] = 2;
60         else myboard[mytop[sortindex[i]] - 1][sortindex[i]] = 1;
61         if (mytop[sortindex[i]] == 1) {
62             mytop[sortindex[i]] = 0;
63         }
64         else if (myboard[mytop[sortindex[i]] - 2][sortindex[i]] == -1) mytop[sortindex[i]] -= 2; //不可落子点, 更新mytop
65     }
66 }

```

```

96 //alpha-beta剪枝
97 if (score > myalpha) {
98     if (mydepth == DEPTH) {
99         next_step[0] = mytop[sortindex[i]] - 1;
100         next_step[1] = sortindex[i];
101     }
102     if (score >= mybeta) {
103         return mybeta;
104     }
105     myalpha = score;
106 }
107
108 delete[] sortindex;
109 return myalpha;
110 }
111

```

(6)、局面的评价函数就是通过计算 top 的 2 子（策略方）和 1 子（用户方）的子的评分，相减。因为影响棋局都多数是 top 的位置，只需计算 top 各子的得分即可。

```

269 //2子点的评价函数
270
271 int evaluate_each_point2(int x, int y, const int M, const int N) { ... }
272
273 //type=true:查看该位置是否必胜棋, type=false:查看威胁棋
274 bool hasThreatPoint(bool type, int x, int y, const int M, const int N) { ... }
275 //棋盘的评价函数
276 int evaluate_board(bool type, const int M, const int N) { ... }

```

## 2、算法改进

### (1)、杀棋的判断



```

312 //考虑横向
313 //计算2子
314 int leftIndex2 = y - 1; //最左端超出2的位置
315 while (leftIndex2 >= 0) {
316     //...
317 }
318 int rightIndex2 = y + 1; //最右端超出2的位置
319 while (rightIndex2 < N) {
320     //...
321 }
322 int horicount2 = rightIndex2 - leftIndex2 - 1; //先算出2的连子数
323
324 int leftIndex22 = leftIndex2;
325 while (leftIndex22 >= 0) {
326     //...
327 }
328 int rightIndex22 = rightIndex2;
329 while (rightIndex22 < N) {
330     //...
331 }
332 int horiwidth2 = rightIndex22 - leftIndex22 - 1; //最大走子范围，再判断是不是活字
333 switch (horicount2) {
334     case 1:
335         if (horiwidth2 > 3) score2 += 2000;
336         else score2 += 500;
337         break;
338     case 2:
339         if (mytop[rightIndex2] == x + 1 && mytop[leftIndex2] == x + 1) { ///判断杀棋走法
340             score2 += 9999;
341             break;
342         }
343         if (horiwidth2 > 3) score2 += 5000;
344         else score2 += 1000;
345         break;
346     case 3:
347         if (mytop[rightIndex2] == x + 1 && mytop[leftIndex2] == x + 1) { ///判断杀棋走法
348             score2 += 99999;
349             break;
350         }
351         if (horiwidth2 > 3) score2 += 10000;
352         else score2 += 2000;
353         break;
354     default:
355         return 99999999;
356 }

```

上图是所谓的杀棋，对局势起到至关重要作用的棋局，给予很大的分数，可以增强评价函数的准确性（原因在之前分析过了）。

## (2)、攻击系数

```

724 bool hasThreatPoint(bool type, int x, int y, const int M, const int N) {
725     //棋盘的评价函数
726     int evaluate_board(bool type, const int M, const int N) {
727         int temp_score2 = 0;
728         int temp_score1 = 0;
729         int xindex, yindex;
730         for (int i = 0; i < N; i++) {
731             if (mytop[i] == M) continue;
732             else {
733                 xindex = mytop[i];
734                 yindex = i;
735                 //if (myboard[xindex][yindex] == 2) temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
736                 //if (myboard[xindex][yindex] == 1) temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
737                 //判断走的该步是不是危险走步
738                 if (myboard[xindex][yindex] == 2) {
739                     temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
740                     if (mytop[i] > 0) {
741                         if (hasThreatPoint(false, xindex - 1, yindex, M, N)) temp_score2 -= 9999999;
742                     }
743                 }
744                 if (myboard[xindex][yindex] == 1) {
745                     temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
746                     if (mytop[i] > 0) {
747                         if (hasThreatPoint(true, xindex - 1, yindex, M, N)) temp_score1 -= 9999999;
748                     }
749                 }
750             }
751         }
752         //////
753         if (type) return (int)(0.5*temp_score2) - temp_score1;
754         else return (temp_score1 - int(0.5*temp_score2));
755     }
756 }

```

红框中的 0.5 是所谓的攻击系数。Temp\_score2 是策略方的得分，系数 0.5 小于 1，表示策略方比较更多的考虑地方的走步，这样的设计可以使得在该评价函数下的走步策略不是那么“刚性”大，比较多的考虑敌方走步，可以弥补评价函数的缺陷。

## (3)、判断最后子节点是不是危险走步

```

796 // 判断是否是危险走步
797 int evaluate_board(bool type, const int M, const int N) {
798     int temp_score2 = 0;
799     int temp_score1 = 0;
800     int xindex, yindex;
801     for (int i = 0; i < N; i++) {
802         if (mytop[i] == M) continue;
803         else {
804             xindex = mytop[i];
805             yindex = i;
806             //if (myboard[xindex][yindex] == 2) temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
807             //if (myboard[xindex][yindex] == 1) temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
808             //判断走的这一步是不是危险走步
809             if (myboard[xindex][yindex] == 2) {
810                 temp_score2 += evaluate_each_point2(xindex, yindex, M, N);
811                 if (mytop[i] > 0) {
812                     if (hasThreatPoint(false, xindex - 1, yindex, M, N)) temp_score2 -= 9999999;
813                 }
814             }
815             if (myboard[xindex][yindex] == 1) {
816                 temp_score1 += evaluate_each_point1(xindex, yindex, M, N);
817                 if (mytop[i] > 0) {
818                     if (hasThreatPoint(true, xindex - 1, yindex, M, N)) temp_score1 -= 9999999;
819                 }
820             }
821             //////
822         }
823     }
824     if (type) return (int)(0.5*temp_score2) - temp_score1;
825     else return (temp_score1 - int(0.5*temp_score2));
826 }

```

如图的在到达中子节点的时候，判断一下是不是危险走步，用于对评价函数进行校正。细细一想，其实相当于考虑的层数更深了。

#### (4)、ab 剪枝的排序算法：提高剪枝效率

```

28 int negmaxsearch(bool mytype, int mydepth, int myalpha, int mybeta, const int M, const int N) {
29     int score=0;
30     //判断局面：
31     if (machineWin(lastx, lasty, M, N, myboard) || userWin(lastx, lasty, M, N, myboard) || mydepth == 0 || isTie(N, mytop))
32         return evaluate_board(mytype, M, N);
33 }
34 //剪枝排序！越靠近lasty, 优先拓展，可以极大提升效率，深度可再加深2层！
35 int* sortindex = new int[N];
36 int count = 0;
37 for (int delta = 0; delta < N; delta++) {
38     if (count == N) break;
39     for (int j = 0; j < N; j++) {
40         if (abs(j - lasty) == delta) {
41             sortindex[count] = j;
42             count++;
43             if (count == N) break;
44         }
45     }
46 }
47 }
48 //节点拓展
49 for (int i = 0; i < N; i++) {
50     if (mytop[sortindex[i]] == 0) continue; //满的列不拓展
51     //备份拓展前的点
52     int backx = lastx;
53     int backy = lasty;
54     int *backtop = new int[N];
55     for (int j = 0; j < N; j++) {
56         backtop[j] = mytop[j];
57     }
58     int **backboard = new int*[N];
59     for (int p = 0; p < M; p++) {
60         backboard[p] = new int[N];

```

其实 ab 剪枝的效率和子节点的排序有关系的。直观的想法，如果先考虑优秀的节点，那么剪枝效果会很好。根据理论推算，最佳剪枝效率可使得每层的等效节点数是全拓展节点数的平方根。也就是，如果本来每层要搜 10000 步，最佳剪枝可以每层等效搜 100 步。效率极大提升。

图中的按照靠近上一步走棋的附近开始拓展，这样可以使原本 6 层的深度，拓展到 8 层。

### 3、测试结果

使用我的策略与测试策略对弈 10 轮，结果如下：

结果（相对策略）	胜	负	平
100	0.45	0.55	0
90	0.6	0.4	0
80	0.7	0.3	0
70	0.95	0.05	0
60	0.85	0.15	0

54	0.9	0.1	0
40	0.95	0.05	0
34	0.9	0.1	0
20	0.9	0.1	0
10	0.9	0.1	0

可见在我设计的评价函数基础下，8 层的搜索深度效果还是非常好的。

#### 4、收获体会

通过调参发现

(1)、评价函数的重要性非常大，评价函数的好坏直接影响策略的性能。所以我的上述改进算法很多是用来弥补评价函数的，效果非常好！

(2)、评价函数、拓展深度、攻击系数三者是相互补充，相互影响的。通过调参发现，有些参数会明显增强性能。这个归根结底还是优化评价函数。

(3)、攻击系数越大，策略刚性越大，越容易在早期获胜，越到晚期，劣势越大。

(4)、ab 剪枝节点排序算法可以非常好的提升剪枝效率。搜索深度从原来 6 层，增加到 8 层。