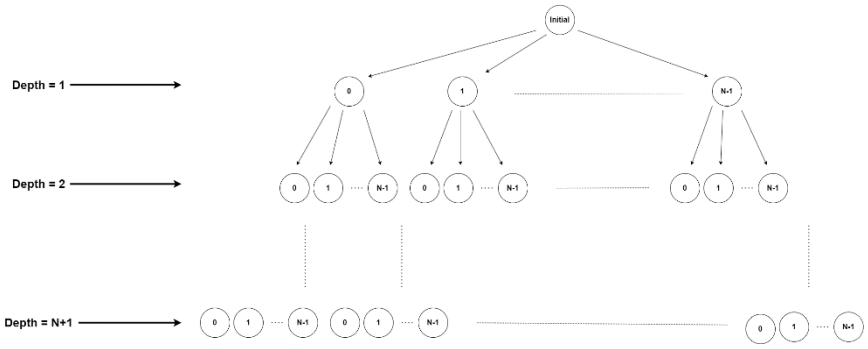
IDS implementation



(圖 1) IDS search tree

IDS 的概念轉變成 N 皇后的棋盤，配合圖 1，代表:
Depth=1 時，嘗試在第 0 個 column 的 0~(N-1) row 放置皇后。
如果沒有 optimal solution 則 Depth+1;
Depth=2 時，嘗試在第 0,1 個 column 的 0~(N-1) row 放置皇后。
如果沒有 optimal solution 則 Depth+1;
…
Depth=N 時，嘗試在第 0,1,…,N-1 個 column 的 0~(N-1) row 放置皇后。
找到 optimal solution，IDS 結束;

```
bool Graph::IDS(int row=0) {
    for (int i = 0; i <= MAX_DEPTH; i++) {
        cout << "Depth: " << i << endl;
        if (DLS(i,row) == true)
            return true;
    }
    return false;
}
```

(圖 2)

左圖 2 代表 Depth 遞增的方程式。

若該 Depth no optimal solution，則 Depth 遞增，直到找到 optimal solution 為止。

```
bool Graph::DLS(int limit,int row) {
    if (limit==0 && find_a_sol())
        return true;

    if (limit <= 0)
        return false;

    for (int col = 0; col < N; col++) {
        if(queens_attack_num(row,col)==0){
            board[row][col] = 1;
            solution[row] = col;
            if(DLS(limit-1,row+1)){
                return true;
            }
            board[row][col] = 0;
            solution[row] = NOT_FOUND;
        }
    }
    return false;
}
```

(圖 3)

左圖 3 代表從 Depth=0 往下 DFS 到 Depth limit 的深度為止。

每加深一層，代表已在上一層 col 放置皇后。

每相同深度層都會 BFS，嘗試在同個 col 上的每個 row 放上皇后，試著尋找 optimal solution。

HC implementation

以下是我的 HC pseudocode:

1. 隨機產生初始棋盤(initial_state)
2. 以初始棋盤(initial_state)為 HC 的出發點，令當前棋盤(current_state) = 初始棋盤(initial_state)
3. 產生當前棋盤(current_state)的 neighbor
4. 比較 neighbor 和 current_state 的 queen_attack_nums:
   If (neighbor 的 queen_attack_nums < current_state 的 queen_attack_nums):
   current_state
   else:
   do nothing
5. 重複 step3、4 直到 max iterations / 找到 optimal solution

```
for(int i=0;i<n;i++){
    initial_state.push_back(i);
}

int swap_times=100;
for (int i=0;i<swap_times;i++) {
    int x = rand() % N;
    int y = rand() % N;
    std::swap(initial_state[x], initial_state[y]);
}
```

左圖 4 的 Initail_state 代表 n*n 大小的棋盤。
HC 能否得出 optimal solution 的關鍵在於"初始點"。
我的初始點是隨機產生的，會隨機 swap 皇后位置 10 次。

(圖 4) 產生初始棋盤

```
vector<int> generate_random_neighbor(vector<int>& state){

    int src = rand()%N;
    int des = rand()%N;

    vector<int> copy_state(state);
    std::swap(copy_state[src],copy_state[des]);

    return copy_state;
}
```

左圖 5 是產生 neighbor 的方式。

方式是隨機挑選 current_state 的兩個皇后互相交換位置。

(圖 5) 產生 neighbor

因為 HC 產生初始點、產生 neighbor 充滿了隨機性，且 solution 受到 initial_state 非常大的影響，HC 很容易卡在 Local optimal solution。稍後的數據研究也發現 max_iterations 調的在大，solution 也很難往最佳解邁進。

GA implementation

Initial population:

隨機產生 5 個 N*N 的 state。

排列方式被隨機打亂。

Fitness:

對於 GA for N 皇后的 fitness，我會計算當前 N*N board 的 queen_attack_nums:

queen_attack_nums 越少，代表接近 optimal solution，fitness 高。

queen_attack_nums 越多，代表遠離 optimal solution，fitness 低。

Retain Elite:

留下菁英的方式非常簡單 -> 尋找當前群體(population)中 fitness 最高的 state。

產生新族群時，我會從舊族群中選出 fitness 最高的兩個 state，這兩個 state 是新族群最初的兩個 state。

Selection:

Selection 有隨機也有競爭。

我會從族群中隨機挑選 x 個 state，從這 x 個 state 裡面挑出最優秀的當作 parent，準備生小孩。

Crossover:

我的 crossover 的切割點在正中間。(總是如此)

child 的基因 = father 前半的基因 + mother 後半的基因 ->

新棋盤 = father 的左半邊 + mother 的右半邊。

Mutation:

隨機挑選基因的兩個位置做 swap -> 隨機挑選棋盤的兩個 col 做 swap。

數據

IDS

Run time 跟 board size 相關。

IDS 總是能到 optimal solution，亦即 8-Queen 和 50-Queen 都能得到#attack 為 0 的解。但是 50-Queen 用 IDS 解需要非常久的時間，本人等了 4 小時左右 Depth 還在第 6 層，離第 50 層非常遙遠，因此數據紀錄用 too large 表示。

IDS			
8-Queen		50-Queen	
#attack	time(second)	#attack	time(second)
0	0.006	?	too large

(圖 6) IDS 8-Queen、50-Queen 數據

HC

HC 的執行時間主要和 max_iterations 掛勾，performance 則不掛勾。

iteration	1000	
8-Queen	Hill Climbing(HC)	
Record	#attack	time(second)
1	1	0.0248
2	0	0.0749
3	1	0.1142
4	1	0.0469
5	1	0.0406
6	1	0.0272
7	0	0.0627
8	1	0.0666
9	1	0.0555
10	1	0.0511
11	1	0.0678
12	1	0.0495
13	0	0.0237
14	0	0.0379
15	1	0.0640
16	1	0.0429
17	0	0.0650
18	0	0.0270
19	0	0.0891
20	1	0.0697
21	1	0.0279
22	0	0.0885
23	1	0.0747
24	1	0.0746
25	1	0.0254
26	1	0.0972
27	0	0.0878
28	1	0.0774
29	1	0.0786
30	0	0.0256
average	0.667	0.059
SR	0.333	

(圖 7) HC 8-Queen

50-Queen	Hill Climbing(HC)							
iteration	10		100		1000		10000	
Record	#attack	time(second)	#attack	time(second)	#attack	time(second)	#attack	time(second)
1	28	0.150	31	0.030	34	0.029194832	25	0.243947983
2	24	0.036	33	0.023	30	0.025125742	25	0.217440128
3	39	0.036	27	0.032	26	0.037302732	29	0.176468134
4	25	0.028	29	0.014	34	0.026805401	27	0.242183924
5	27	0.036	33	0.097	24	0.041974545	32	0.220527649
6	26	0.037	26	0.052	31	0.038707018	34	0.189006567
7	27	0.051	28	0.051	30	0.048197985	21	0.191246748
8	37	0.048	36	0.037	24	0.036665201	24	0.186606884
9	32	0.037	28	0.037	21	0.029037237	30	0.189015388
10	36	0.036	26	0.033	29	0.029085636	28	0.189415216
11	44	0.034	31	0.028	20	0.032001495	30	0.189177275
12	37	0.036	33	0.016	43	0.030119658	35	0.256973028
13	25	0.035	32	0.024	23	0.025493383	32	0.208775043
14	31	0.034	25	0.038	25	0.025701284	34	0.230618238
15	28	0.046	27	0.024	30	0.035196781	28	0.276524305
16	28	0.036	30	0.024	26	0.027062416	29	0.18152523
17	27	0.038	31	0.059	30	0.030210733	28	0.185530663
18	24	0.020	28	0.046	30	0.029413939	23	0.187886477
19	30	0.050	27	0.026	34	0.031496048	34	0.250317812
20	39	0.035	17	0.026	24	0.03700304	21	0.205910444
21	35	0.041	22	0.040	31	0.030159473	29	0.20112586
22	37	0.035	26	0.045	35	0.038102865	31	0.191459417
23	21	0.053	33	0.036	29	0.034116268	32	0.238562346
24	33	0.039	28	0.050	33	0.098756075	37	0.412290096
25	30	0.014	25	0.047	28	0.027146101	28	0.184317112
26	28	0.045	21	0.055	27	0.027065516	27	0.179665804
27	23	0.042	30	0.038	34	0.030841112	30	0.216056108
28	24	0.078	26	0.032	28	0.031824827	24	0.193530083
29	29	0.016	29	0.028	31	0.034694672	26	0.285685778
30	39	0.015	31	0.026	28	0.03030014	26	0.199107409
average	30.433	0.041	28.300	0.037	29.067	0.034	28.633	0.217
SR	0.000		0.000		0.000		0.000	

(圖 8) HC 50-Queen

圖 7 可以看到 8-Queen 的 HC 解 average #attack 是 0.667，SR 為 0.333，成功率不到一半大約在 1/3。

圖 8 可以看到 50-Queen 的 HC 解配合各種 max_iteration 的 average #attack 和 SR。

For SR: 可以看到 max_iteration 提高對於 SR 沒有顯著增加，是因為 HC 作為 local search，非常吃 initial_state 的資訊，當 initial_state 落點不好，HC 容易卡在 Local optimal solution。這邊棋盤為 50*50，很難有好的初始落點，因此攻擊數都很高。

For average #attack: Average #attack 約在 28~30。與 SR 同裡，當 initial_state 落點不好，HC 容易卡在 Local optimal，平均攻擊數不因為 max_iteration 遞增而改變。

GA

我的 GA 有三個參數:

max_generation = 族群最大世代

size_of_population = 族人最多數目

mutation_probability = 基因變異機率

分別代表底下欄位的 generation、population、mutation。

Average Run time 跟 **max_generation**、**size_of_population** 相關。

generation	1000	
population	50	
mutation	10%	
8-Queen	Genetic Algorithm(GA)	
Record	#attack	time(second)
1	0	0.397
2	0	0.431
3	0	0.438
4	0	0.569
5	0	0.474
6	1	0.431
7	0	0.368
8	0	0.892
9	0	0.53
10	1	0.43
11	0	0.352
12	1	0.483
13	0	0.655
14	3	0.537
15	0	0.53
16	0	0.487
17	0	0.38
18	0	0.474
19	1	0.408
20	0	0.49
21	0	0.463
22	0	0.403
23	0	0.497
24	1	0.487
25	0	0.499
26	0	0.582
27	0	0.499
28	0	0.475
29	0	0.684
30	1	0.5
average	0.3	0.495
SR	21/30	

(圖 9) GA 8-Queen

50-Queen	Genetic Algorithm(GA)							
generation	10		100		1000		10000	
population	50		50		50		50	
mutation	10%		10%		10%		10%	
Record	#attack	time(second)	#attack	time(second)	#attack	time(second)	#attack	time(second)
1	35	0.036	6	0.214	3	1.568	0	14.325
2	40	0.059	9	0.239	4	1.474	4	14.101
3	33	0.058	6	0.228	4	2.338	1	13.963
4	28	0.095	4	0.206	3	2.050	1	14.144
5	26	0.062	8	0.284	1	1.669	4	14.863
6	21	0.093	9	0.233	3	1.515	3	14.464
7	25	0.085	7	0.294	2	1.474	1	14.088
8	22	0.034	7	0.252	3	1.467	4	13.968
9	19	0.072	6	0.210	1	1.462	3	13.722
10	32	0.033	9	0.215	4	1.439	4	14.108
11	36	0.030	9	0.234	2	1.440	3	13.684
12	27	0.040	10	0.206	0	1.583	5	14.004
13	29	0.046	7	0.208	2	1.398	4	14.290
14	30	0.045	6	0.255	4	1.468	4	13.821
15	37	0.045	7	0.209	1	1.611	1	13.143
16	27	0.062	7	0.230	1	1.536	4	13.050
17	25	0.042	5	0.208	0	1.488	6	16.319
18	25	0.044	4	0.222	4	1.461	4	17.143
19	22	0.034	7	0.226	2	1.539	1	13.996
20	35	0.039	9	0.207	5	1.461	4	14.199
21	32	0.074	4	0.203	0	1.527	3	13.203
22	22	0.030	5	0.210	4	1.467	3	14.050
23	22	0.084	11	0.412	0	1.454	5	13.865
24	29	0.126	9	0.471	1	1.479	6	14.026
25	30	0.070	3	0.376	3	1.471	3	13.031
26	48	0.033	7	0.250	3	1.533	3	14.238
27	32	0.036	6	0.213	0	1.422	2	13.299
28	28	0.059	7	0.240	3	1.482	8	15.328
29	39	0.042	7	0.320	2	1.422	5	14.319
30	20	0.043	7	0.229	2	1.412	4	15.526
average	29.200	0.055	6.933	0.250	2.233	1.537	3.433	14.209
SR	0.000		0.000		0.167		0.033	

(圖 10) GA 50-Queen。Generation 遞增

圖 9 可以看到 8-Queen 的 GA 解 average #attack 是 0.3，SR 為 21/30，generation=1000、population=50、mutation=10%，效果比 HC 好很多。

圖 10 可以看到 50-Queen 的 GA 解，且 generation 遞增(多世代)。

For SR: 這次數據沒得出 optimal solution，但有多個#attack=1 的數據，我認為只是跑 30 次還不夠多，次數拉高一定會有#attack=0 出現。

For average #attack: 可以看到 generation 提高對於 SR 有不錯的進步。我的分析是每次產生 new generation 時，會從上世代挑出兩個最佳基因當作初始基因，因此當世代數拉高時，好的基因都會被保留，且被挑出的兩個最佳基因會越變越好。但在 generation=10000 時 average #attack 提高，我推測是 selection 時選到不好的 parent 交配，影響了群體的表現。

50-Queen	Genetic Algorithm(GA)							
generation	10		10		10		10	
population	10		100		1000		10000	
mutation	10%		10%		10%		10%	
Record	#attack	time(second)	#attack	time(second)	#attack	time(second)	#attack	time(second)
1	162	0.033	13	0.054805756	10	0.379762888	7	4.006943226
2	190	0.048	23	0.092349768	10	0.386160612	8	3.554322958
3	155	0.023	25	0.101688147	14	0.539426088	10	3.166448116
4	158	0.046	24	0.097766161	8	0.372961283	8	3.457111597
5	190	0.013	20	0.104404688	13	0.381053448	7	3.672307968
6	152	0.040	14	0.082841873	9	0.322854996	6	3.435698271
7	157	0.038	23	0.099436283	13	1.121277094	7	3.351773739
8	204	0.041	18	0.107525587	7	0.454102039	5	3.235574484
9	199	0.032	18	0.113605738	11	0.484360218	7	3.726983547
10	170	0.040	15	0.048428535	11	0.674549341	7	3.116618156
11	160	0.040	13	0.119575262	10	0.952589512	7	3.180073261
12	175	0.021	19	0.072383404	6	0.538106918	11	3.059312344
13	156	0.037	28	0.099668026	12	0.39756608	7	3.02353406
14	175	0.014	21	0.0751369	7	0.415460348	4	3.01774621
15	130	0.048	21	0.122510195	7	0.546302557	10	2.999605179
16	185	0.021	23	0.053402901	12	0.669145346	8	3.159022808
17	131	0.020	19	0.159623146	13	0.622535706	7	3.042049408
18	191	0.044	18	0.089529753	11	0.438628674	6	2.995072603
19	116	0.039	21	0.142059803	11	0.654447079	6	3.074400663
20	228	0.041	18	0.046545506	13	0.479051828	10	3.056968212
21	158	0.015	22	0.065428734	12	0.50528121	10	3.053391218
22	120	0.014	16	0.09189415	15	0.438655376	11	3.246603966
23	186	0.028	16	0.164579391	10	0.494062662	8	3.049911261
24	146	0.043	20	0.151112318	10	0.441095114	8	3.127658844
25	202	0.038	16	0.128031969	10	0.421112299	7	3.026942492
26	142	0.046	17	0.100792408	12	0.373751163	7	3.118513584
27	196	0.039	17	0.154033899	11	0.501826763	8	3.294769526
28	168	0.039	21	0.15050149	11	0.477242947	6	3.038766146
29	176	0.038	21	0.124586344	9	0.618960619	12	3.183728933
30	130	0.038	14	0.10750103	10	0.252965689	9	3.181350708
average	166.933	0.034	19.133	0.104	10.600	0.512	7.800	3.222
SR	0.000		0.000		0.000		0.000	

(圖 11) GA 50-Queen。population 遞增

圖 10 可以看到 50-Queen 的 GA 解，且 population 遞增(多族人)。
可以看到 average #attack 越來越小，但 SR 仍為 0。

圖 10 與圖 11 比較下來，我得出的資訊是:

- Population 固定、Generation 遞增，即使 population 比較少，但容易留下更好的基因，average #attack 更低。
- Population 遞增、Generation 固定，基因也在進步，但我的 selectiong 是隨機挑人，在從中挑最好的，也會挑到不好的基因，因此這邊的 average #attack 沒有圖 10 的數據漂亮。