

A woman with dark hair tied back is shown in profile, smelling a red flower. A robotic arm with a camera eye is holding the flower. The background is a gradient of blue and red.

# PEG SOLITAIRE AI SOLVER

Artificial Intelligence Software Program Development



## Nodes

The nodes are defined as the basic storage units in the Algorithm. In the base code, it is a struct formation which initially includes integer variable "depth", enumeration type variable "move", state struct type variable "state", and node struct pointer "parent" for its parent nodes. In order to free all nodes, which were pushed in the stack, in the last phase of the solving the Peg Solitaire puzzle, the pointer node struct pointer "next\_free" was added in the node struct. It is worth to mention that move enumeration type includes the directions of the legal movement while the state struct contains the current Peg Solitaire composition, the position of the cursor, and the selection state for saving the final solutions.

```
typedef struct {
    int8_t x,y;
} position_s;

// player controlling
typedef struct {
    int8_t field[SIZE][SIZE];
    position_s cursor;
    bool selected;
} state_t;

/**
 * Move type
 */
typedef enum moves_e{
    left=0,
    right=1,
    up=2,
    down=3
} move_t;

struct node_s{
    int depth;
    move_t move;
    state_t state;
    // we add a node pointer for the
    struct node_s* next_free;
    struct node_s* parent;
};

typedef struct node_s node_t;
```

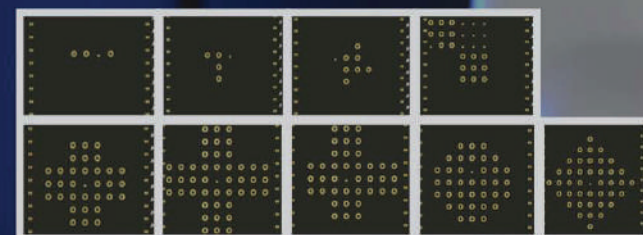
## Introduction

The primary purpose of this project is to explore the method based on the principle of the AlphaGo to solve the Peg Solitaire problem. That is, the nodes pointed their parent nodes will be created to store all possibilities of the current Peg Solitaire composition to form a graphic which has many connected nodes. Whilst the Depth First Search (DFS) strategy is implemented to find out the deepest path generated in the graph.

Experiments shows that the AI is able to solve the first 6 layouts provided by the instruction entirely. In the statistic of solving the sixth layouts of the Peg Solitaire problem, the 4898609 nodes were generated, and 1090275 of them were popped out from the stack for analysing. The problem was solved in 6.031 seconds.

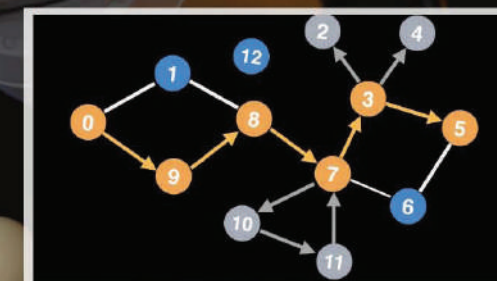
## Depth First Search (DFS)

Depth First Search (DFS) is an algorithm that is implemented to traverse a connected graph. The mechanism is that the algorithm starts at one original vertex V0, and the algorithm tries to traverse all the vertex to the end. If the computer finds that it cannot reach the goals, it will return to the previous node, and then tries to find another path to reach the destination. The conception of this algorithm is trying to achieve the vertex as deep as it can (2020 Geeksforgeeks). In the AI solver from this paper, this DFS was used to optimise and save the solutions while the algorithm is executing.



## The layouts of Peg Solitaire

After observation, we can see that difficulties of the puzzles get more challenging as the name code of layouts increases. That is, the simplest one is the first layout which just has three pegs and four slots on the board, in contrary, the most difficult one is the 9th layout that has 40 pegs and 41 slots on the board. According to the conceptions of mathematic, the increase of pegs and slots means the growth of the entropy. In other words, by increasing the number of pegs and slots, the possibilities of the current Peg Solitaire composition will increase, which could increase the space complexity and time complexity when AI solves the puzzles.





In the process of optimisation, according to the principle that the most Peg Solitaire boards are symmetrical, the rotation of Peg Solitaire board has been introduced into the code. During the comparison of new nodes, the algorithm will continuously rotate Peg Solitaire board into a different direction (Graph 6) to determine if current Peg Solitaire composition is in the hash table. This preliminary optimisation could significantly reduce the frequency of storing duplicate nodes to prevent massive unnecessary calculations. In terms of time complexity, the author increased the  $O(4^n) = O(n)$  time complexity for adding the rotations of the boards but reduced the  $O(n \cdot \log(n))$  time complexity caused by the AI solver diving into the duplicate solving paths.

[illegible]

CODE: [github.com/chenjiang0819/AlSolverForPegSolitaire-GraphSearch](https://github.com/chenjiang0819/AlSolverForPegSolitaire-GraphSearch)

The hash table is used to filter the possibilities that have already existed in the stack. When the AI solver generates a new node, the algorithm will compare the Peg Solitaire composition of the new node and the Peg Solitaire compositions in the hash table. The AI solver will add it in the hash table and push the new nodes into the stack if the Peg Solitaire composition does not exist in the hash table. Otherwise, the new node is going to be discarded. Thus, the Peg Solitaire compositions of the nodes in the stack and the Peg Solitaire compositions existed in the hash table will be synchronised on time.

It is worth to mention that the size of the hash table is dynamic. That is, the size of the hash table increases with the number of possibilities generated so that it can hold thousands of Peg Solitaire compositions. On the other hand, the searching key of the hash table is produced by a designed hash function which converts a Peg Solitaire composition input into a string. This design ensures both accurate searching and sufficient memory capacity. It also gives a direction to optimal the AI solver.

In the AI solver, the stack is used to store the nodes that contain possibilities of the next step. In each “while” cycle, one node will be popped out to analyse all possibilities of the next step and execute actions. After the previous step, new results generated by the actions will be counted as the possibilities and are separately stored in the new nodes. If Peg Solitaire compositions of the new nodes(possibilities) do not exist in the stack, the new nodes will be pointed to its parent nodes and pushed into the stack. Moreover, in order to prevent memory leaks caused by nodes loss, and free all nodes in the stack in the last stage of solving the puzzle, the linked list structures are created to store the nodes that put in the same position in the stack. From the aspect of the process, it is not hard to see that AI solver implement the DFS by simply using the stack type structure.

After experiments, we can conclude that given 1.5 million calculated budget, the maximum of solving capacity of the AI solver is up to the 6th layout - Layout 5 (36 pegs game). We can see that as the number of pegs increases, the amount of computation increases exponentially, which was because more possibilities had been generated. In the report, it is quite difficult to precisely estimate the complexity of increasing one peg, because there were too many elements that would affect the calculation amount. However, after comparing different budget inputs and results of the Layout 5 (36 pegs game), Layout 6 (44 pegs game), Layout 7 (38 pegs game), and Layout 8 (40 pegs game), we can say that if AI solver digs through one layer of the solution path, 10-20 times as many nodes as the previous layer generated will be produced. This implies that if one peg was added on the board, the complexity of solving this problem would be increased  $O(10^{(\text{length of the solution})})$ . Therefore, reducing possibilities (new nodes) produced by algorithm could significantly improve the AI solver. Moreover, the shape of Peg Solitaire board could have a significant effect on the efficiency of the AI solver. In the experiments, the author found that asymmetrical boards are more AI solver friendly. This phenomenon suggests that the hash table and its hash function, which used to filter the duplicate possibility, may need to be further improved.

In addition, from the results of the Layout 5 (36 pegs game), the Layout 6 (44 pegs game), Layout 7 (38 pegs game), and Layout 8 (40 pegs game), we could also find out that once the AI solver finished the calculation of one of the solution layers, the AI solver would not turn back to that layer to generate new possibilities. In other words, with the increase of computation budget and computation time, solution lengths are continuously increased. In the other hand, this mechanism brings a potential problem. On the result of the Layout 7 (38 pegs game), from budget 100K to budget 5.4M, the AI solver was stuck in the 36th layer of the solution path. That is,  $98.1\% (1 - (100K/5.4M)) = 98.1\%$  budget were invested on finding the correct path to break through the 36th layer of solution, and then finally solved the puzzle. Therefore, introducing the technics of machine learning such as deep learning, convolutional neural network, and k-means, which could predict the next few steps of the current Peg Solitaire composition, will be significant meaningful for this kind of the AI solver.

```

004 int ht_insertOrUpdate(table, void* key, void* value) {
005     struct ht_node*
006         node;
007     struct ht_node**
008         nodes;
009     struct ht_node**
010         nodes2;
011     struct ht_node**
012         nodes3;
013     struct ht_node**
014         nodes4;
015     struct ht_node**
016         nodes5;
017     struct ht_node**
018         nodes6;
019     struct ht_node**
020         nodes7;
021     struct ht_node**
022         nodes8;
023     struct ht_node**
024         nodes9;
025     struct ht_node**
026         nodes10;
027     struct ht_node**
028         nodes11;
029     struct ht_node**
030         nodes12;
031     struct ht_node**
032         nodes13;
033     struct ht_node**
034         nodes14;
035     struct ht_node**
036         nodes15;
037     struct ht_node**
038         nodes16;
039     struct ht_node**
040         nodes17;
041     struct ht_node**
042         nodes18;
043     struct ht_node**
044         nodes19;
045     struct ht_node**
046         nodes20;
047     struct ht_node**
048         nodes21;
049     struct ht_node**
050         nodes22;
051     struct ht_node**
052         nodes23;
053     struct ht_node**
054         nodes24;
055     struct ht_node**
056         nodes25;
057     struct ht_node**
058         nodes26;
059     struct ht_node**
060         nodes27;
061     struct ht_node**
062         nodes28;
063     struct ht_node**
064         nodes29;
065     struct ht_node**
066         nodes30;
067     struct ht_node**
068         nodes31;
069     struct ht_node**
070         nodes32;
071     struct ht_node**
072         nodes33;
073     struct ht_node**
074         nodes34;
075     struct ht_node**
076         nodes35;
077     struct ht_node**
078         nodes36;
079     struct ht_node**
080         nodes37;
081     struct ht_node**
082         nodes38;
083     struct ht_node**
084         nodes39;
085     struct ht_node**
086         nodes40;
087     struct ht_node**
088         nodes41;
089     struct ht_node**
090         nodes42;
091     struct ht_node**
092         nodes43;
093     struct ht_node**
094         nodes44;
095     struct ht_node**
096         nodes45;
097     struct ht_node**
098         nodes46;
099     struct ht_node**
100         nodes47;
101     struct ht_node**
102         nodes48;
103     struct ht_node**
104         nodes49;
105     struct ht_node**
106         nodes50;
107     struct ht_node**
108         nodes51;
109     struct ht_node**
110         nodes52;
111     struct ht_node**
112         nodes53;
113     struct ht_node**
114         nodes54;
115     struct ht_node**
116         nodes55;
117     struct ht_node**
118         nodes56;
119     struct ht_node**
120         nodes57;
121     struct ht_node**
122         nodes58;
123     struct ht_node**
124         nodes59;
125     struct ht_node**
126         nodes60;
127     struct ht_node**
128         nodes61;
129     struct ht_node**
130         nodes62;
131     struct ht_node**
132         nodes63;
133     struct ht_node**
134         nodes64;
135     struct ht_node**
136         nodes65;
137     struct ht_node**
138         nodes66;
139     struct ht_node**
140         nodes67;
141     struct ht_node**
142         nodes68;
143     struct ht_node**
144         nodes69;
145     struct ht_node**
146         nodes70;
147     struct ht_node**
148         nodes71;
149     struct ht_node**
150         nodes72;
151     struct ht_node**
152         nodes73;
153     struct ht_node**
154         nodes74;
155     struct ht_node**
156         nodes75;
157     struct ht_node**
158         nodes76;
159     struct ht_node**
160         nodes77;
161     struct ht_node**
162         nodes78;
163     struct ht_node**
164         nodes79;
165     struct ht_node**
166         nodes80;
167     struct ht_node**
168         nodes81;
169     struct ht_node**
170         nodes82;
171     struct ht_node**
172         nodes83;
173     struct ht_node**
174         nodes84;
175     struct ht_node**
176         nodes85;
177     struct ht_node**
178         nodes86;
179     struct ht_node**
180         nodes87;
181     struct ht_node**
182         nodes88;
183     struct ht_node**
184         nodes89;
185     struct ht_node**
186         nodes90;
187     struct ht_node**
188         nodes91;
189     struct ht_node**
190         nodes92;
191     struct ht_node**
192         nodes93;
193     struct ht_node**
194         nodes94;
195     struct ht_node**
196         nodes95;
197     struct ht_node**
198         nodes96;
199     struct ht_node**
200         nodes97;
201     struct ht_node**
202         nodes98;
203     struct ht_node**
204         nodes99;
205     struct ht_node**
206         nodes100;
207     struct ht_node**
208         nodes101;
209     struct ht_node**
210         nodes102;
211     struct ht_node**
212         nodes103;
213     struct ht_node**
214         nodes104;
215     struct ht_node**
216         nodes105;
217     struct ht_node**
218         nodes106;
219     struct ht_node**
220         nodes107;
221     struct ht_node**
222         nodes108;
223     struct ht_node**
224         nodes109;
225     struct ht_node**
226         nodes110;
227     struct ht_node**
228         nodes111;
229     struct ht_node**
230         nodes112;
231     struct ht_node**
232         nodes113;
233     struct ht_node**
234         nodes114;
235     struct ht_node**
236         nodes115;
237     struct ht_node**
238         nodes116;
239     struct ht_node**
240         nodes117;
241     struct ht_node**
242         nodes118;
243     struct ht_node**
244         nodes119;
245     struct ht_node**
246         nodes120;
247     struct ht_node**
248         nodes121;
249     struct ht_node**
250         nodes122;
251     struct ht_node**
252         nodes123;
253     struct ht_node**
254         nodes124;
255     struct ht_node**
256         nodes125;
257     struct ht_node**
258         nodes126;
259     struct ht_node**
260         nodes127;
261     struct ht_node**
262         nodes128;
263     struct ht_node**
264         nodes129;
265     struct ht_node**
266         nodes130;
267     struct ht_node**
268         nodes131;
269     struct ht_node**
270         nodes132;
271     struct ht_node**
272         nodes133;
273     struct ht_node**
274         nodes134;
275     struct ht_node**
276         nodes135;
277     struct ht_node**
278         nodes136;
279     struct ht_node**
280         nodes137;
281     struct ht_node**
282         nodes138;
283     struct ht_node**
284         nodes139;
285     struct ht_node**
286         nodes140;
287     struct ht_node**
288         nodes141;
289     struct ht_node**
290         nodes142;
291     struct ht_node**
292         nodes143;
293     struct ht_node**
294         nodes144;
295     struct ht_node**
296         nodes145;
297     struct ht_node**
298         nodes146;
299     struct ht_node**
300         nodes147;
301     struct ht_node**
302         nodes148;
303     struct ht_node**
304         nodes149;
305     struct ht_node**
306         nodes150;
307     struct ht_node**
308         nodes151;
309     struct ht_node**
310         nodes152;
311     struct ht_node**
312         nodes153;
313     struct ht_node**
314         nodes154;
315     struct ht_node**
316         nodes155;
317     struct ht_node**
318         nodes156;
319     struct ht_node**
320         nodes157;
321     struct ht_node**
322         nodes158;
323     struct ht_node**
324         nodes159;
325     struct ht_node**
326         nodes160;
327     struct ht_node**
328         nodes161;
329     struct ht_node**
330         nodes162;
331     struct ht_node**
332         nodes163;
333     struct ht_node**
334         nodes164;
335     struct ht_node**
336         nodes165;
337     struct ht_node**
338         nodes166;
339     struct ht_node**
340         nodes167;
341     struct ht_node**
342         nodes168;
343     struct ht_node**
344         nodes169;
345     struct ht_node**
346         nodes170;
347     struct ht_node**
348         nodes171;
349     struct ht_node**
350         nodes172;
351     struct ht_node**
352         nodes173;
353     struct ht_node**
354         nodes174;
355     struct ht_node**
356         nodes175;
357     struct ht_node**
358         nodes176;
359     struct ht_node**
360         nodes177;
361     struct ht_node**
362         nodes178;
363     struct ht_node**
364         nodes179;
365     struct ht_node**
366         nodes180;
367     struct ht_node**
368         nodes181;
369     struct ht_node**
370         nodes182;
371     struct ht_node**
372         nodes183;
373     struct ht_node**
374         nodes184;
375     struct ht_node**
376         nodes185;
377     struct ht_node**
378         nodes186;
379     struct ht_node**
380         nodes187;
381     struct ht_node**
382         nodes188;
383     struct ht_node**
384         nodes189;
385     struct ht_node**
386         nodes190;
387     struct ht_node**
388         nodes191;
389     struct ht_node**
390         nodes192;
391     struct ht_node**
392         nodes193;
393     struct ht_node**
394         nodes194;
395     struct ht_node**
396         nodes195;
397     struct ht_node**
398         nodes196;
399     struct ht_node**
400         nodes197;
401     struct ht_node**
402         nodes198;
403     struct ht_node**
404         nodes199;
405     struct ht_node**
406         nodes200;
407     struct ht_node**
408         nodes201;
409     struct ht_node**
410         nodes202;
411     struct ht_node**
412         nodes203;
413     struct ht_node**
414         nodes204;
415     struct ht_node**
416         nodes205;
417     struct ht_node**
418         nodes206;
419     struct ht_node**
420         nodes207;
421     struct ht_node**
422         nodes208;
423     struct ht_node**
424         nodes209;
425     struct ht_node**
426         nodes210;
427     struct ht_node**
428         nodes211;
429     struct ht_node**
430         nodes212;
431     struct ht_node**
432         nodes213;
433     struct ht_node**
434         nodes214;
435     struct ht_node**
436         nodes215;
437     struct ht_node**
438         nodes216;
439     struct ht_node**
440         nodes217;
441     struct ht_node**
442         nodes218;
443     struct ht_node**
444         nodes219;
445     struct ht
```

```
void stack_push(node_t* n) {
    if(stack_top_idx == STACK_SIZE - 1) { // overflow
        printf("Error: stack overflow\n");
        return;
    }
    //In order to free all nodes, we make a linked
    stack_top_idx++;
    if(stack_top_idx != NULL){
        n->next_free = stack_top_idx;
    }
    stack[stack_top_idx] = n;
}

void stack_pop() {
    if(stack_top_idx == -1) { // if stack is empty
        printf("Error: No element to pop\n");
        return;
    }
    stack_top_idx--;
}

node_t* stack_top() {
    return stack[stack_top_idx];
}
```