人工智慧概論

0613413 蔡怡君

－實驗目的：knight in a 8x8 chessboard  
Knight的移動只能(+-1,+-2) or (+-2,+-1) 給定兩點find optimal path (minimum number of steps) path

－實驗方法：(graph-search)  
－使用的概念：在explored set和frontier都沒有相同state的node ( 直接使用table 去記錄這個點有沒有被visit過 )：將其加入frontier !

1. BFS：使用Frontier: **FIFO queue**能找到最佳解
2. DFS：使用Frontier: **FIFO stack** 可以找到解答但不一定會找到最佳解
3. IDS：iteratively run DLS，使用Frontier: **FIFO stack** 能找到最佳解，借用BFS的optimal(第一個找到為最佳解)，再加上DFS的優點：占用空間小，但是expand的node數目比較多。  
   --- 透過Evaluation function去選擇最佳的node去expand ---
4. A\*：使用Frontier: **priority queue**，expand the lowest f(n)的node，能找到最佳解
5. IDA\*：一開始使用的是Frontier: **priority queue**，但透過討論區發現其實應該是使用**stack**，所以有寫兩個版本的IDA\*，再問題區有比較兩者的差別，都能找到最佳解。

－結果觀察：  
**Time:**

**Time成長率：**

|  |  |
| --- | --- |
| BFS | 時間大約與n成正比。 |
| DFS | 時間大約與n^2成正比。 |
| IDS | 隨著n的成長，Time的成長速度也越大。成長幅度比DFS還要大。(看60 ->120) |
| A\* | 時間大約與n成正比。 |
| IDA\* | 時間大約與n成正比。 |

**Node expansion:**

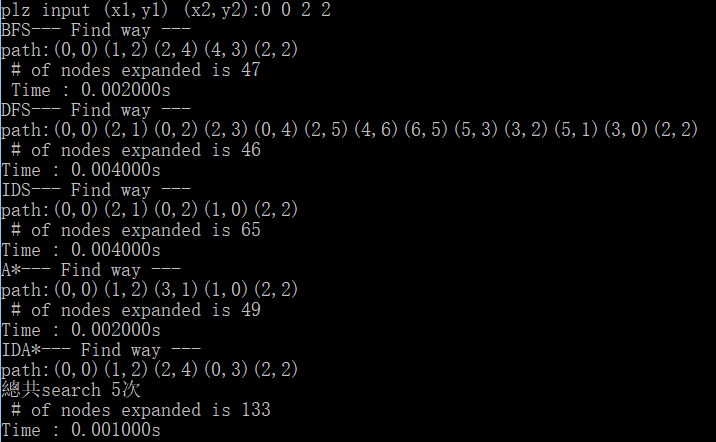
**Node expansion 成長率：**

|  |  |
| --- | --- |
| BFS | 時間大約與n^2成正比。(大約3~4倍) |
| DFS | 時間大約與n^2成正比。(大約4倍) |
| IDS | 時間大約與n^4成正比。(大約15~20倍) |
| A\* | 時間大約與n^2成正比。(大約3~4倍) |
| IDA\* | 時間大約與n^2成正比。(大約3~4倍) |

**最佳解觀察**：

利用老師給的(0,0)->(2,2)，最佳Path為總共經過五個點，觀察答案發現只有DFS沒有最佳的解，其他BFS、IDS、A\*、IDA\*，都能得到最佳解！或是透過其他點，也可以得

到同樣的結果。



－結果分析：

以**時間率**來看，DFS、IDS為表現最壞的兩者，兩者的時間成長率明顯大於其他三個，再以**最佳解**來看DFS雖然會找到解但不一定是最佳解，所以在這個問題上DFS較不適合，再看到node expansion的表在其中IDS為最高的，因為其不斷的加深depth去找到最佳解，所以會展開眾多node，但是它的node占用記憶體不會項BFS這麼多，因為IDS每一次的iteration都會重新restart不是全部存下來。

所以目前看到**時間成長率**與**node expansion成長率**，IDA\*與A\*與BFS為表現較佳者的，但是BFS的space complexity會比其他兩者還要大，因為它會展開不必要的node，像是哩，的space complexity是以指數型成長。最後以IDA\*與A\*來看，兩者都是因為會透過Evaluation function f(n) = g(n) + h(n) 去篩選排除表現較不佳的node，所以可以得知這個Evaluation function在此問題中是consistent的，很有用的節省了時間、node expansion。**所以再這個實驗當中IDA\*跟A\*是比較推薦使用的。**

實驗中的遇到的問題：

**Question 1** : **IDS沒有辦法找到最佳解？**  
**Answer 1 :** 因為我的algorithm是使用「**state並不在explored set，但是已經在frontier就丟棄的條件的話**」，在IDS 中好像會出現 IDS 找不到最佳解的狀況。假設最佳路徑為P1,其經過k node，但透過IDS卻找不到最佳解，是因為程式方面上，先跑Path P2，而P2上面也有經過k node，所以k node在找到最佳解P1前就被丟入frontier，所以在跑最佳解(k點前)的時候，因為k node已經存在於frontier & explored set，所以沒辦法找到最佳解。

**解決方法**：紀錄每一個node的depth，當在產生相同state的時候，若新的深度小於之前記住的，就重新把其加入Frontier，因為如果原本的depth是最小的話，應該要可以找到最佳解。  
  
**Question 2 : 使用priority queue的IDA\*會比使用stack的IDA\*好嗎？**  
經過兩次測值(0,0)->(15,94)：node expansion、time都是priority queue比較好，但是(0,0)->(99,99)發現又變成stack表現較佳。所以這部分還沒有準確的答案可以知道使用哪個方法較佳。

|  |  |
| --- | --- |
| 程式選項5 IDA\* (using priority queue) | 程式選項6 IDA\*(using stack) |
| (0,0) -> (15,94) | (0,0) -> (99,99) |
|  |  |

Thing I have learned：  
這次的作業，可以應用在很多的手機益智遊戲方面，像是8－puzzle，或是下象棋，原來這些遊戲都是可以透過以前演算法所學過的BFS、DFS去解題，還有新教的IDS、A\*、IDA\*，還有新的概念像是evaluation function、heuristic function，去讓一個問題的解變的更明確，減少不必要的計算量、加快得到解答的速度。

Remaining Question：  
因為我在這次的演算法當中是使用的概念是：「在explored set和frontier都沒有相同state的node ( 直接使用table 去記錄這個點有沒有被visit過 )：將其加入frontier 」，所以再Qustion1，有指出IDS、IDA\*要去多檢查之前visit過的深度有沒有比新visit的大，所以很好奇是不是其它兩種狀況的效能會比較高呢？(討論區關於DFS那篇裡面，我也跟他們遇到一樣的問題)。

1. // 程式碼  
   #include<iostream>
2. #include<stdio.h>
3. #include<stdlib.h>
4. #include<math.h>
5. #include<queue>
6. #include<stack>
7. #include<vector>
8. #include<time.h>
9. #define N 100
10. **using** **namespace** std;
12. /\*
13. author : 0613413 蔡怡君
14. content: knight in a 8\*8 chessboard
15. move only allow (+-1,+-2)or(+-2,+-1)
16. find optimal path from 2 given locations
17. \*/
19. **struct** point
20. {
21. **int** x,y;
22. **int** cost; // use for heuristic function
23. }p1,p2,dir[8];
25. // parent 紀錄這一個node上面的爸爸是誰
26. // table 紀錄這一個點有沒有被visited過
27. // low cost using for IDA\* 紀錄目前最小的cost
28. point parent[N][N];
29. **int** table[N][N]={0},lowcost=1,node=1;
31. **void** initial()
32. {
33. dir[0]={1,2};
34. dir[1]={1,-2};
35. dir[2]={-1,2};
36. dir[3]={-1,-2};
37. dir[4]={2,1};
38. dir[5]={2,-1};
39. dir[6]={-2,1};
40. dir[7]={-2,-1};
41. }
43. **void** clear()
44. {
45. // reset table & parent
46. node = 0;
47. **for**(**int** i = 0 ; i < N ; i++){
48. **for**(**int** j = 0 ; j < N ; j++){
49. table[i][j] = 0;
50. parent[i][j].x = -1;
51. parent[i][j].y = -1;
52. }
53. }

56. }
58. **void** print\_path(point now)
59. {
60. **if**(parent[now.x][now.y].x == now.x && parent[now.x][now.y].y == now.y)
61. {
62. cout<<"("<<now.x<<","<<now.y<<")";
63. **return**;
64. }
65. **else**
66. print\_path(parent[now.x][now.y]);
67. cout<<"("<<now.x<<","<<now.y<<")";
68. }
70. **bool** operator < (**const** point &p1,**const** point &p2){ **return** p1.cost < p2.cost;}
71. **bool** operator > (**const** point &p1,**const** point &p2){ **return** p1.cost > p2.cost;}
73. **int** BFS(point p1,point p2)
74. {
75. node = 1;
76. queue<point>front;
77. front.push(p1);
78. table[p1.x][p1.y]=1;
79. parent[p1.x][p1.y]=p1; // root的parent=自己
80. **while**(!front.empty())
81. {
82. point current = front.front();
83. front.pop();
84. // Look 8 directions and checked whether it's visited ?
85. **for**(**int** i = 0 ; i < 8 ; i++){
86. point now = { current.x + dir[i].x , current.y + dir[i].y };
87. **if**(now.x < 0 || now.y < 0 || now.x > N-1 || now.y > N-1) // out of index
88. **continue**;
89. **if**(table[now.x][now.y] == 0) // mean 還沒被走過
90. {
91. front.push(now);
92. node++;
93. table[now.x][now.y] = 1;
94. parent[now.x][now.y] = current;
95. // if we find point! print path and break.
96. **if**(now.x == p2.x && now.y == p2.y)
97. {
98. cout<<"--- Find way ---\npath:";
99. print\_path(front.back());
100. **return** node;
101. }
102. }
103. }
104. }
105. }
107. // 找到 node 時檢查explored set & frontier (單純使用 table去記這點是否被visited)
108. **int** DFS(point p1,point p2)
109. {
110. **int** i,depth[N][N] = {0}; node = 1;
111. stack<point>front;
112. front.push(p1);
113. table[p1.x][p1.y] = 1;
114. parent[p1.x][p1.y] = p1; // root的parent=自己
115. **while**(!front.empty())
116. {
117. point current = front.top();
118. front.pop();
119. //printf("(%d,%d)",current.x,current.y);
120. **for**(i = 0 ; i < 8 ; i ++){
121. point now = { current.x + dir[i].x , current.y + dir[i].y};
122. **if**(now.x < 0 || now.y < 0 || now.x > N-1 || now.y > N-1)
123. **continue**;
124. **if**(table[now.x][now.y] && depth[now.x][now.y] > depth[current.x][current.y]+1)
125. {
126. table[now.x][now.y]=0;
127. }
128. **if**( table[now.x][now.y] == 0)
129. {
130. table[now.x][now.y] = 1;
131. front.push(now);
132. parent[now.x][now.y] = current;
133. depth[now.x][now.y] = depth[current.x][current.y] + 1;
134. node++;
135. **if**(now.x == p2.x && now.y == p2.y) // means arrive
136. {
137. cout<<"--- Find way ---\npath:";
138. print\_path(front.top());
139. **return** node;
140. }
141. }
142. }
143. }
144. **return** 0;
145. }
147. // -- Depth-limited Search
148. **int** DLS(point p1,point p2,**int** d)
149. {
150. // add depth[][] to memorize depth of the node
151. **int** i, depth[N][N]={0};
152. node = 1;
153. stack<point>front;
154. front.push(p1);
155. table[p1.x][p1.y]=1;
156. parent[p1.x][p1.y] = p1; // root的parent=自己
157. //printf("\n\ndepth limit:%d\n",d);
158. **while**(!front.empty())
159. {
160. point current = front.top();
161. front.pop();
162. // control depth
163. **if**(depth[current.x][current.y] == d)
164. **continue**;
165. **for**(i = 0 ; i < 8 ; i ++){
166. point now = { current.x + dir[i].x , current.y + dir[i].y};
167. **if**(now.x < 0 || now.y < 0 || now.x > N-1 || now.y > N-1)
168. **continue**;
169. // 新的點 之前被走過但之前的深度比較大 現今走的depth比較小 所以要再丟入frontier一次
170. **if**(table[now.x][now.y] && depth[now.x][now.y] > depth[current.x][current.y]+1)
171. {
172. table[now.x][now.y]=0;
173. }
174. **if**( table[now.x][now.y] == 0)
175. {
176. table[now.x][now.y] = 1;
177. front.push(now);
178. parent[now.x][now.y] = current;
179. depth[now.x][now.y] = depth[current.x][current.y]+1; // depth = 原來的點 +1
180. node++;
181. **if**(now.x == p2.x && now.y == p2.y) // means arrive
182. {
183. cout<<"--- Find way ---\npath:";
184. print\_path(front.top());
185. **return** node;
186. }
187. // printf("depth=%d. (%d,%d)",depth[now.x][now.y],now.x,now.y);
188. }
189. }
190. }
191. **return** 0; // if it's not found return 0
192. }
194. // iterative-deepening search
195. **int** IDS(point p1,point p2)
196. {
197. // i for control depth
198. **int** result = 0, i = 1 ,lowcost = 1, sum =0;
199. **while**(1){
200. clear();
201. result = DLS(p1,p2,i++);
202. sum += node;
203. **if**(result != 0)
204. **return** sum;
205. }
206. **return** 0;
207. }
209. **int** h(point n,point p2)
210. {
211. **int** x,y;
212. x = (n.x > p2.x) ? n.x - p2.x : p2.x - n.x;
213. y = (n.y > p2.x) ? n.y - p2.y : p2.y - n.y;
214. **if**(n.x == p2.x && n.y == p2.y)
215. {
216. x=-6666; y=-6666;
217. }
218. **return** floor((x+y)/3);
219. }
221. // A\*
222. **int** Astar(point p1,point p2)
223. {
224. **int** depth[N][N] = {0}; node=1;
225. priority\_queue<point,vector<point>,greater<point> >front;
226. front.push(p1);
227. table[p1.x][p1.y]=1;
228. parent[p1.x][p1.y]=p1;
229. **while**(!front.empty())
230. {
231. point current = front.top();
232. front.pop();
233. **for**(**int** i = 0 ; i < 8 ; i ++){
234. point now = { current.x + dir[i].x, current.y + dir[i].y};
235. **if**(now.x < 0 || now.y < 0 || now.x > N-1 || now.y > N-1)
236. **continue**;
237. **if**(table[now.x][now.y] == 0)
238. {
239. table[now.x][now.y] = 1;
240. depth[now.x][now.y] = depth[current.x][current.y] + 1;
241. parent[now.x][now.y] = current;
242. node ++;
243. now.cost = depth[now.x][now.y] + h(now,p2);
244. front.push(now);
245. //printf("depth=%d. (%d,%d) cost=%d\n",depth[now.x][now.y],now.x,now.y,now.cost);
246. **if** ( now.x == p2.x && now.y == p2.y )
247. {
248. cout<<"--- Find way ---\npath:";
249. print\_path(front.top());
250. **return** node;
251. }
252. //cout<<"now.cost"<<depth[now.x][now.y]<<"+"<<h(now,p2)<<"="<<now.cost<<endl;
253. }
254. }
255. }
256. }
258. // using for IDA\*
259. // Deepening Limited A\*
261. **int** DLAstar(point p1,point p2)
262. {
263. // cost = 第一次跑DLAstar的 low cost;
264. **int** depth[N][N] = {0},cost = lowcost;
265. node = 1;
266. **bool** flag = **false**;
268. priority\_queue<point,vector<point>,greater<point> >front;
269. front.push(p1);
270. table[p1.x][p1.y]=1;
271. parent[p1.x][p1.y]=p1;
272. **while**(!front.empty())
273. {
274. point current = front.top();
275. front.pop();
276. **for**(**int** i = 0 ; i < 8 ; i ++){
277. point now = { current.x + dir[i].x, current.y + dir[i].y};
278. **if**(now.x < 0 || now.y < 0 || now.x > N-1 || now.y > N-1)
279. **continue**;
280. **if**(table[now.x][now.y] == 0)
281. {
282. table[now.x][now.y] = 1;
283. depth[now.x][now.y] = depth[current.x][current.y] + 1;
284. parent[now.x][now.y] = current;
285. now.cost = depth[now.x][now.y] + h(now,p2);
286. node ++;
287. // push 之前要看這個點的 f() 是否超過 limited f()？
288. **if**(now.cost > cost)
289. {
290. // 紀錄第一次 大於low cost 的 flag 要用於下一次
291. **if**(!flag){ lowcost = now.cost; flag = **true**; }
292. // cout<<"the step lowcost:"<<lowcost<<endl;}
293. //cout<<"超過limit 不push進去 -";
294. }
295. **else**
296. front.push(now);
297. //printf("(%d,%d) cost=%d\n",now.x,now.y,now.cost);
298. **if** ( now.x == p2.x && now.y == p2.y )
299. {
300. cout<<"--- Find way ---\npath:";
301. print\_path(front.top());
302. **return** node;
303. }
304. }
305. }
306. }
307. **return** 0;
308. }
310. **int** IDAstar(point p1,point p2)
311. {
312. **int** result = 0, i = 1, sum =0;
313. lowcost = 1;
314. **while**(1)
315. {
316. clear();
317. result = DLAstar(p1,p2);
318. sum += node; i++;
319. //cout<<"第"<<i++<<"次try"<<"lower cost:"<<lowcost<<endl;
320. **if**(result != 0)
321. {
322. cout<<"\n總共search "<<i<<"次";
323. **return** sum;
324. }
325. }
327. }

330. **int** DLAstar\_stack(point p1,point p2)
331. {
332. // cost = 第一次跑DLAstar的 low cost;
333. **int** depth[N][N] = {0},cost = lowcost;
334. node = 1;
335. **bool** flag = **false**;
337. stack<point>front;
338. front.push(p1);
339. table[p1.x][p1.y]=1;
340. parent[p1.x][p1.y]=p1;
341. **while**(!front.empty())
342. {
343. point current = front.top();
344. front.pop();
345. **for**(**int** i = 0 ; i < 8 ; i ++){
346. point now = { current.x + dir[i].x, current.y + dir[i].y};
347. **if**(now.x < 0 || now.y < 0 || now.x > N-1 || now.y > N-1)
348. **continue**;
349. // 新的點 之前被走過但之前的深度比較大 現今走的depth比較小 所以要再丟入frontier一次
350. **if**(table[now.x][now.y] && depth[now.x][now.y] > depth[current.x][current.y]+1)
351. table[now.x][now.y]=0;
352. **if**(table[now.x][now.y] == 0)
353. {
354. table[now.x][now.y] = 1;
355. depth[now.x][now.y] = depth[current.x][current.y] + 1;
356. parent[now.x][now.y] = current;
357. now.cost = depth[now.x][now.y] + h(now,p2);
358. node ++;
359. // push 之前要看這個點的 f() 是否超過 limited f()？
360. **if**(now.cost > cost)
361. {
362. // 紀錄第一次 大於low cost 的 flag 要用於下一次
363. **if**(!flag){ lowcost = now.cost; flag = **true**;  }
364. //cout<<"the step lowcost:"<<lowcost<<endl;
365. //cout<<"超過limit 不push進去 -";
366. }
367. **else**
368. front.push(now);
369. //printf("(%d,%d) cost=%d\n",now.x,now.y,now.cost);
370. **if** ( now.x == p2.x && now.y == p2.y )
371. {
372. cout<<"--- Find way ---\npath:";
373. print\_path(front.top());
374. **return** node;
375. }
376. }
377. }
378. }
379. **return** 0;
380. }
382. **int** IDAstar\_stack(point p1,point p2)
383. {
384. **int** result = 0, i = 1, sum =0;
385. lowcost = 1;
386. **while**(1)
387. {
388. clear();
389. result = DLAstar\_stack(p1,p2);
390. sum += node; i++;
391. //cout<<"第"<<i++<<"次try"<<"lower cost:"<<lowcost<<endl;
392. **if**(result != 0)
393. {
394. cout<<"\n總共search "<<i<<"次";
395. **return** sum;
396. }
397. }
399. }

402. // only using for 印 node/ time
403. **void** check(point p1,point p2)
404. {
405. **time\_t** start,end;
406. **double** t;
407. cout<<"BFS";
408. start = clock();
409. cout<<"\n # of nodes expanded is "<<BFS(p1,p2)<<endl;
410. end = clock();
411. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
412. printf(" Time : %fs\n",t);
413. clear();
415. cout<<"DFS";
416. start = clock();
417. cout<<"\n # of nodes expanded is "<<DFS(p1,p2)<<endl;
418. end = clock();
419. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
420. printf("Time : %fs\n",t);
421. clear();
423. cout<<"IDS";
424. start = clock();
425. cout<<"\n # of nodes expanded is "<<IDS(p1,p2)<<endl;
426. end = clock();
427. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
428. printf("Time : %fs\n",t);
429. clear();
431. cout<<"A\*";
432. start = clock();
433. cout<<"\n # of nodes expanded is "<<Astar(p1,p2)<<endl;
434. end = clock();
435. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
436. printf("Time : %fs\n",t);
437. clear();
439. cout<<"IDA\*";
440. start = clock();
441. cout<<"\n # of nodes expanded is "<<IDAstar(p1,p2)<<endl;
442. end = clock();
443. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
444. printf("Time : %fs\n",t);
445. clear();
446. }
448. **int** main()
449. {
450. **int** type;
451. initial();
452. cout<<"--- Welcome to 8-kight ---\n";
453. **time\_t** start,end;
454. **double** t;
455. **while**(1)
456. {
457. cout<<"\n\nchioce 1:BFS 2:DFS 3:IDS 4:A\* 5:IDA\* 6.IDA\* (using stack)\n";
458. cout<<"plz input type of algorithm:";
459. cin>>type;
460. cout<<"plz input (x1,y1) (x2,y2):";
461. cin>>p1.x>>p1.y>>p2.x>>p2.y;
462. clear();
463. //check(p1,p2);
464. **switch**(type)
465. {
466. **case** 1:
467. cout<<"BFS";
468. start = clock();
469. cout<<"\n # of nodes expanded is "<<BFS(p1,p2)<<endl;
470. end = clock();
471. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
472. printf(" Time : %fs",t);
473. **break**;
474. **case** 2:
475. cout<<"DFS";
476. start = clock();
477. cout<<"\n # of nodes expanded is "<<DFS(p1,p2)<<endl;
478. end = clock();
479. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
480. printf("Time : %fs",t);
481. **break**;
482. **case** 3:
483. cout<<"IDS";
484. start = clock();
485. cout<<"\n # of nodes expanded is "<<IDS(p1,p2)<<endl;
486. end = clock();
487. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
488. printf("Time : %fs",t);
489. **break**;
490. **case** 4:
491. cout<<"A\*";
492. start = clock();
493. cout<<"\n # of nodes expanded is "<<Astar(p1,p2)<<endl;
494. end = clock();
495. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
496. printf("Time : %fs",t);
497. **break**;
498. **case** 5:
499. cout<<"IDA\*";
500. start = clock();
501. cout<<"\n # of nodes expanded is "<<IDAstar(p1,p2)<<endl;
502. end = clock();
503. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
504. printf("Time : %fs",t);
505. **break**;
506. **case** 6:
507. cout<<"IDA\* using stack";
508. start = clock();
509. cout<<"\n # of nodes expanded is "<<IDAstar\_stack(p1,p2)<<endl;
510. end = clock();
511. t = ((**double**)(end-start))/CLOCKS\_PER\_SEC;
512. printf("Time : %fs",t);
513. **break**;
514. **default**:
515. cout<<"----- warning:wrong input -----\n";
516. }
517. }
518. **return** 0;
519. }