دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

هوش مصنوعی
پاییز ۱۳۹۹
استاد: محمدحسین رهبان

پاسخ تمرین اول، بخش دوم — **جست‌وجوی آگاهانه و توابع اکتشافی** — مهلت ارسال: ـ

سوالات نظری (۴۰ نمره)

۱. (۱۵ نمره)

(آ) فرض کنید که مینیمم تعداد حرکت لازم برای این که عدد i در حالت کنونی (state) به سرجایش برگردد، $d_{i,state}$ باشد. تابع heuristic را به این صورت تعریف می‌کنیم: $h_{state} = max_i d_{i,state}$. واضح است که این تابع admissible است. هم‌چنین این تابع monotonic است. زیرا با هر شیفت دوری، هر $d_{i,state}$ حداکثر یکی کم می‌شود. بنابراین با هر حرکت، تابع heuristic حداکثر یکی کم می‌شود. پس داریم : $h_a \leq ab + h_b$. ضمنا واضح است که این تابع $\Theta(n)$ مقدار مختلف اختیار می‌کند.

.

(ب) دوباره همانند قسمت اول، فرض کنید که مینیمم تعداد حرکت لازم برای این که عدد i در حالت کنونی (state) به سرجایش برگردد، $d_{i,state}$ باشد. تابع heuristic را به این صورت تعریف می‌کنیم:

$$h_{state} = \frac{\sum_{i=1}^{n^2} d_{i,state}}{n}$$

. می‌دانیم که d عددی از $O(n)$ است. هم‌چنین در مجموع n^2 عدد داریم. پس می‌توانیم بگوییم که h_{state} عددی از $O(n^2)$ است.(دقت کنیم که حالتی که ضریبی از n^2 باشد نیز وجود دارد). هر شیفت دوری، حداکثر یک واحد از d اعداد درون سطر یا ستون کم می‌کند. پس h_{state} حداکثر یک واحد کم می‌شود. بنابراین می‌دانیم این تابع monotonic است. از آنجا که حداکثر مقدار این تابع از $\Theta(n^2)$ بود و هر مرحله هم حداکثر ۱ واحد کم می‌شود و هنگامی که به جواب برسیم به ۰ میرسد، $\Theta(n^2)$ مقدار متمایز دارد.

۲. (۵ نمره) در الگوریتم *IDA تعداد دفعاتی که الگوریتم DFS را صدا می‌کنیم، بستگی به مقادیر مختلفی دارد که تابع f اختیار می‌کند. اگر در گرافی که میخواهیم مساله‌ی فروشنده‌ی دوره‌گرد را حل کنیم،
n راس داشته باشیم، و وزن‌های یال ها و هم‌چنین تابع heuristic جوری باشند که f، n مقدار مختلف را اختیار کند، باید n بار الگوریتم DFS را اجرا کنیم. ضمنا در دفعه‌ی ،ام-i i راس expand می‌شوند(i راسی که تابع f شان کمینه است). در نتیجه تعداد کل عملیات ها از $O(n^2)$ خواهد بود. پس اگر n عدد بزرگی باشد، راه حل ما زیاد طول می‌کشد و مناسب نیست. اما در مسئله‌ی 8-puzzle ، تعداد f های مختلف کم است و در نتیجه ما تعداد دفعاتی که الگوریتم DFS را صدا می‌زنیم کم است. به همین دلیل، تعداد راس‌هایی که در این روش expand می‌شوند، حدودا برابر تعداد راس‌هایی است که در الگوریتم *A ، expand می‌شوند. ضمن این که این جا دیگر نیازی به priority queue نداریم و الگوریتم DFS را با استک پیاده سازی می‌کنیم. پس الگوریتم *IDA ، برای مسئله‌ی 8-puzzle بهتر است.

۳. (۸ نمره) این جمله غلط است. فرض کنید که یک گراف ۳ راسی داشته باشیم و یال های بین رئوس به شکل زیر باشد(e_{ij} به معنای یال از راس i به راس j است).

$$e_{12} = e_{23} = e_{31} = -1$$

ادعا می‌کنیم که الگوریتم در این حالت، پایان پذیر نیست. فرض کنید که از راس ۱ الگوریتم را اجرا کنیم. در گام اول، فاصله‌ی راس ۲ برابر ۱− قرار داده می‌شود. سپس فاصله‌ی راس ۳ برابر ۲− قرار داده می‌شود. در گام بعدی، فاصله‌ی راس ۱ برابر ۳− می‌شود. چون با یک فاصله‌ی منفی به راس اولیه برگشتیم، می‌توانیم فاصله‌های سایر راس ها را مدام کمتر کنیم. در واقع اگر در گرافمان دور منفی داشته باشیم، الگوریتم دیگر پایان پذیر نخواهد بود و فرض این که اولین باری که یک راس را expand می‌کنیم حتما به کمینه‌ی فاصله‌ی راس مذکور دست‌یافته‌ایم غلط خواهد بود. همچنین اگر از graph search استفاده کنیم، با استفاده از دور منفی و ایده‌ی مشابه، می‌توان گرافی ارائه کرد که الگوریتم در زمان نمایی اجرا می‌شود.

۴. (۱۲ نمره)

- درمورد توابع $h۱$ و $h۲$

(آ) $h۱$ یک تابع consistent است. $h۲$ تابعی consistent نیست. زیرا داریم :

$$h۲[B] = ۱۲ > BC + h۲[C] = ۱ + ۱۰ = ۱۱$$

(ب) در هر دو مسیر پیموده شده به شکل زیر است: $A \to C \to D \to G$ که هزینه این مسیر برابر است با ۱۶.

(ج) در هر دو مسیر پیموده شده به شکل زیر است: $A \to B \to C \to D \to F \to G$ که مسیر بهینه است و هزینه‌ش برابر است با ۱۳.

- برای $h۳$:

(آ) consistency برای admissible بودن، بازه‌ی معتبر، برابر $[۱۲،۰]$ است.

(ب) اگر مقدار از ۹ کمتر اکید باشد، یال AB دیگر consistent نخواهد بود. برای consistent بودن یال BC نیز نمی‌تواند از ۱۰ بزرگتر باشد. بقیه‌ی یال‌ها نیز اگر عدد در بازه‌ی $[۹, ۱۰]$ باشند، شرط consistent بودنشان درست است. پس بزرگترین بازه $[۹, ۱۰]$ است.

(ج) برای این که اول راس C دیده شود، مقدار heuristic باید بیشتر از ۱۲ باشد. اما می‌دانیم که $A \to B$ از مسیر $A \to C \to B$ وزن کمتری دارد و امکان ندارد راس B توسط راس C باز شود. پس نحوه‌ی باز شدن راس‌ها به شکل $A \to C \to B \to D$ ممکن نیست. پس بازه‌ی مورد نظر، تهی است.

سوالات عملی (۴۰ نمره)

۱. (۴۰ نمره) در این سوال، شما باید هر یک از حالت‌های ممکن روبیک را یک استیت در نظر بگیرید و دو استیت به هم دیگر یال دارند، اگر با شیفت یکی از سطرها یا ستون‌ها به اندازه‌ی یک واحد(در یکی از جهت‌ها)، از یک استیت به استیت دیگر برسیم. در گراف ساخته شده، واضح است که وزن هر یال برابر ۱ است و مسیر با کمینه یال از استیت کنونی به استیت‌نهایی، جواب مسئله است. در نتیجه ما باید از یک الگوریتم shortest path استفاده کنیم. از آنجا که در سوال یک بخش تئوری، دو تابع heuristic تعریف کرده‌ایم، می‌توانیم هر یک از آنها را استفاده کنیم و با استفاده از الگوریتم A* جواب مسئله را بدست آوریم. در قسمت زیر نیز، کد آقای ابوالفضل اسد را برای این سوال مشاهده می‌کنید.

```
#include <iostream>
#include <vector>
#include <stack>
#include <queue>

using namespace std;

int RUBIC_3X3_DIS[] = {1, 1, 0, 1, 1};
int *PTR_3X3 = &RUBIC_3X3_DIS[2];
```

```
10
11 int RUBIC_4X4_DIS[] = {1, 2, 1, 0, 1, 2, 1};
12 int *PTR_4X4 = &RUBIC_4X4_DIS[3];
13
14 int n;
15
16 class Rubic {
17     public:
18         int mat[4][4];
19         int heuristic;
20         int path;
21         Rubic(int mat[][4]) {
22             for (int i = 0; i < n; ++i)
23                 for (int j = 0; j < n; ++j)
24                     this->mat[i][j] = mat[i][j];
25             path = 0;
26             setHeuristic();
27         }
28         Rubic(Rubic* rub, int next) {
29             path = rub->path + 1;
30             for (int i = 0; i < n; ++i)
31                 for (int j = 0; j < n; ++j)
32                     mat[i][j] = rub->mat[i][j];
33             switch (next / n) {
34             case 0:
35                 right(rub, next % n);
36                 break;
37             case 1:
38                 left(rub, next % n);
39                 break;
40             case 2:
41                 up(rub, next % n);
42                 break;
43             case 3:
44                 down(rub, next % n);
45                 break;
46             }
47             setHeuristic();
48         }
49         void right(Rubic* rub, int row) {
50             for (int j = n; j < n * 2; ++j)
51                 mat[row][j % n] = rub->mat[row][(j - 1) % n];
52         }
53         void left(Rubic* rub, int row) {
54             for (int j = n; j < n * 2; ++j)
55                 mat[row][j % n] = rub->mat[row][(j + 1) % n];
56         }
57         void up(Rubic* rub, int col) {
58             for (int i = n; i < n * 2; ++i)
59                 mat[i % n][col] = rub->mat[(i + 1) % n][col];
60         }
61         void down(Rubic* rub, int col) {
62             for (int i = n; i < n * 2; ++i)
63                 mat[i % n][col] = rub->mat[(i - 1) % n][col];
64         }
65
66         void setHeuristic() {
67             heuristic = 2;
```

```
            for (int i = 0; i < n; ++i)
                for (int j = 0; j < n; ++j) {
                    int num = mat[i][j];
                    int r = num / n;
                    int c = num % n;
                    if (n == 4) {
                        heuristic += PTR_4X4[r - i];
                        heuristic += PTR_4X4[c - j];
                    } else {
                        heuristic += PTR_3X3[r - i];
                        heuristic += PTR_3X3[c - j];
                    }

                }
            heuristic /= n;
        }

        void pr() {
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    cout << mat[i][j] << "\t";
                }
                cout << endl;
            }
            cout << endl;

            heuristic = 0;
            for (int i = 0; i < n; ++i)
                for (int j = 0; j < n; ++j) {
                    int num = mat[i][j];
                    int r = num / n;
                    int c = num % n;
                    if (n == 4) {
                        heuristic += PTR_4X4[r - i];
                        heuristic += PTR_4X4[c - j];
                    } else {
                        heuristic += PTR_3X3[r - i];
                        heuristic += PTR_3X3[c - j];
                    }
                    cout << "(" << i << "  " << j << ")" << endl;
                    cout << "r: " << r << endl;
                    cout << "c: " << c << endl;
                    cout << "hue: " << heuristic << endl << endl;
                }
            heuristic /= n;
        }
};

int main() {
    int mat[4][4];
    cin >> n;

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            cin >> mat[i][j];
            mat[i][j]--;
        }

```

```
126     auto cmp = [](Rubic* left, Rubic* right) {
127         return left->path + left->heuristic > right->path + right->
    heuristic;
128     };
129     priority_queue<Rubic*, vector<Rubic*>, decltype(cmp)> stack(cmp);
130     stack.push(new Rubic(mat));
131 
132     while (!stack.empty()) {
133         Rubic* rub = stack.top();
134         stack.pop();
135         if (rub->heuristic == 0) {
136             cout << rub->path << endl;
137             // rub->pr();
138             break;
139         }
140         for (int i = 0; i < n * 4; ++i) {
141             stack.push(new Rubic(rub, i));
142         }
143         delete rub;
144     }
145 
146 }
```