دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

هوش مصنوعی

پاییز ۱۳۹۹
استاد: محمدحسین رهبان

پاسخ تمرین اول، بخش اول — مقدمه و جست‌وجو با هزینه‌ی یکنواخت — مهلت ارسال: ــ

سوالات نظری (۴۰ نمره)

۱. (۸ نمره)

	Fully observable	Single agent	Deterministic	Episodic	Discrete
ربات بازی‌کننده پینگ پونگ	No	No	yes	No	No
عامل هوشمند پشت بازی پوکر	No	No	No	No	Yes
دستگاه تشخیص چهره	No	Yes	Yes	No	Yes
ماشین خودران	No	Yes/No	Yes	No	No

در این بخش، تعریف episodic این درنظر گرفته شده است که استیت کنونی مستقل از استیت قبلی نباشد. برای مثال در رانندگی، سرعت و شتاب تحت تاثیر استیت قبلی است و مستقل نیست.

۲. (۱۰ نمره)

۳. (۱۲ نمره)

- درست
اگر در الگوریتم BFS، goal بودن یک راس را در زمانی بررسی کنیم که بخواهیم آن را expand کنیم، تعداد راس های باز شده در بدترین حالت الگوریتم BFS بیشتر از IDS است. هم‌چنین راس‌های بیشتری را نیز در حافظه نگه می‌داریم. پس استفاده از IDS بهتر است. اما اگر هنگامی که یک راس را expand می‌کنیم، همان جا بررسی کنیم که بچه‌ها goal هستند یا خیر، زمان BFS بهتر می‌شود ولی هم‌چنان حافظه IDS بهتر است.
- غلط
اگر راس تکراری نداشته باشیم، graph search همانند BFS عمل می‌کند و می‌دانیم که IDS که یک tree search است از BFS بهتر عمل می‌کند(راس‌های کمتری باز می‌کند). پس نمی‌توان گفت که همیشه graph search بهتر tree search عمل می‌کند حتی اگر محدودیت حافظه نداشته باشیم.
- غلط
فرض کنید که رئوس همگی غیر تکراری باشند، عمق جواب ۳ باشد و برای رسیدن به جواب، باید از ریشه به دومین فرزندش برویم. در الگوریتم IDS، هنگامی که از ریشه به بچه‌ی اول می‌رویم، دیگر در هیچ زمانی بچه‌ی دوم ریشه نمی‌رسیم. زیرا بی‌نهایت مسیر دیگر وجود دارد که در آنها از ریشه به بچه‌ی اول می‌رویم و IDS در آن مسیرها گیر می‌افتد. بنابراین در این حالت از state ها، الگوریتم IDS کامل نیست.

- درست
در الگوریتم depth-limit search تعداد راس های باز شده برابر است با $۱ + b + \cdots + b^d$ و تعداد راس‌های در مموری هم برابر است با db. اما در الگوریتم BFS حداقل تعداد راس‌های باز شده برابر است با $۱ + b + \cdots + b^d$ و همچنین تعداد راس‌های درون حافظه نیز برابر است با b^d. بنابراین بهتر است که همیشه از الگوریتم depth-limit-search استفاده کنیم.

۴. (۱۰ نمره)

(آ) ابتدا مقدار میانگین را برای BFS محاسبه می‌کنیم. می‌دانیم در هر صورت، $۱ + b + \cdots + b^d$ راس دیده می‌شود تا به عمق d برسیم. حال اگر راس goal، راس ام-i در رئوس در عمق d باشد که expand می‌شوند، $b * (i - ۱)$ راس از عمق $d + ۱$ دیده می‌شود. پس میانگین تعداد رئوس برابر است با میانگین راس هایی که از عمق $d + ۱$ می‌بینیم و رئوس عمق ۰ تا d:

$$E = ۱ + b + \cdots + b^d + \frac{b(۰ + ۱ + \cdots + (b^d - ۱))}{b^d} = ۱ + b + \cdots + b^d + \frac{b(b^d(b^d - ۱))}{(b^d * ۲)}$$
$$= ۱ + b + \cdots + b^d + \frac{b(b^d - ۱)}{۲}$$

حال میانگین را برای DFS محاسبه می‌کنیم. فرض کنید که الگوریتم DFS هنگامی که به راس goal میرسد، دیگر expand نمی‌شود ولی الگوریتم تمام نمی‌شود و الگوریتم مابقی راس‌ها را می‌بیند. حال اگر ترتیب راس‌هایی که ما در الگوریتم DFS دیده‌ایم به شکل $v_۱, v_۲, \cdots, v_k$ باشد، می‌دانیم که $v_k, v_{k-۱}, \cdots, v_۱$ نیز یک ترتیب معتبر دیگر برای DFS است. هم‌چنین فرض کنید که v_i راس goal باشد. می‌دانیم در ترتیب اول، الگوریتم اصلی رئوس $v_۱, v_۲, \cdots, v_i$ را می‌بیند. هم چنین در ترتیب دوم، الگوریتم رئوس $v_k, v_{k-۱}, \cdots, v_i$ را می‌بیند. پس ما اگر هر ترتیب از رئوس مثل $v_۱, v_۲, \cdots, v_k$ را با $v_k, v_{k-۱}, \cdots, v_۱$ متناظر کنیم، می‌دانیم در مجموع، الگوریتم برای هر جفت متناظر شده، $k + ۱$ راس دیده است و تمامی ترتیب های دیدن رئوس نیز هم‌شانس هستند. پس می‌توان گفت که تعداد راس های میانگین که در اجرای الگوریتم DFS می‌بینیم برابر است با $\frac{k+۱}{۲}$. حال کافی است که k را بدست آوریم.
k برابر تمام رئوس درخت جستجواست به جز رئوس زیر درخت راس هدف. بنابراین k برابر است با :

$$۱ + b + \cdots + b^m - (b + b^۲ + \cdots + b^{m-d})$$

(ب) $b = ۲, d = ۳, m = ۴$

زمان اجرا در الگوریتم BFS:

$$۱ + ۲ + ۴ + ۸ + \frac{۲(۷)}{۲} = ۲۲$$

زمان اجرا در الگوریتم DFS:

$$\frac{۱ + ۲ + ۴ + ۸ + ۱۶ - (۲) + ۱}{۲} = ۱۵$$

$$b = ۳, d = ۵, m = ۵$$

زمان اجرا در الگوریتم BFS:

$$۱ + ۳ + ۹ + ۲۷ + ۸۱ + ۲۴۳ + \frac{۳(۲۴۲)}{۲} = ۷۲۷$$

زمان اجرا در الگوریتم DFS:

$$\frac{۱ + ۳ + ۹ + ۲۷ + ۸۱ + ۲۴۳ + ۱}{۲} = \frac{۳۶۵}{۲}$$

(ج) با توجه به روابط بدست آمده در قسمت های قبل، می‌فهمیم که اگر $d << m$ باشد بهتر است که از BFS استفاده کنیم. در صورتی که $m - d$ برابر ۰ باشد، می‌توان فهمید که به ازای درخت های یکسان، الگوریتم DFS بهتر خواهد بود. همچنین برای حالاتی که $m - d$ حداکثر ۱ باشد، الگوریتم DFS همچنان بهتر است. و برای سایر حالات BFS بهتر است.

سوالات عملی (۲۰ نمره)

۱. (۲۰ نمره) فرض کنید که وزن یال‌های مسیری کمینه از a به b را برابر ۰ کرده‌ایم. در این صورت، یا مسیر کمینه بین c و d هیچ یال مشترکی با این مسیر ندارد، یا در راس x به مسیر با یال ۰ وارد می‌شود و در راس y از مسیر با یال ۰ خارج می‌شود. همچنین می‌دانیم که x اولین راس مشترک دو مسیر است و y آخرین راس مشترک است و کمینه مسیر بین x و y متشکل از یال‌های با وزن ۰ مسیر کمینه‌ی بین a و b است. در این صورت مسیر کمینه بین c و d مسیری به شکل $c \to x \to y \to d$ خواهد بود. پس اگر $dis(i, j)$ مینیمم فاصله‌ی راس i و j باشد، کمینه مسیر بین c و d برابر است با : $dis(c, x) + dis(y, d)$.
حال کافی است به ازای هر x که در مسیر کمینه‌ی بین a و b است، راسی را بیابیم که آن هم در این مسیر کمینه باشد و فاصله اش با راس d کمینه باشد. این راس را به این شکل تعریف می‌کنیم: $BestCase[x] = y$.
می‌دانیم که به ازای هر x ، راس $BestCase[x]$ راسی است که $dis(BestCase[x], b) \leq dis(x, b)$.
همچنین می‌دانیم که $BestCase[b] = b$. پس اگر فاصله‌ی هر راسی را از راس‌های a, b, c, d بدانیم، کافی است که برای هر راس(که در مسیر کمینه‌ی بین a و b هستند)، راس $BestCase$ را بیابیم. ادعا می‌کنیم که این مقادیر را با الگوریتمی مشابه UCS بدست آوریم. به این شکل که اگر از راس b شروع کنیم و الگوریتم را اجرا کنیم ، برای راس x که به تازگی باز شده است، $BestCase$ برابر است با خودش، یا $BestCase$ همسایه‌هایش مثل t که $dis(x, b) = edge(x, t) + dis(t, b)$. و از بین تمام گزینه‌های موجود، راسی را بر می‌گزینیم که فاصله‌ش تا راس d کمینه باشد. بنابراین $BestCase$ تمامی راس ها بدست می‌آید و می‌توانیم جواب را محاسبه کنیم. کد این الگوریتم را نیز در زیر مشاهده می‌کنید.

```
#include<cstdio>
#include<vector>
#include<queue>
#include<algorithm>

using namespace std;

typedef pair<long long, int> P;

const long long inf = 1e18;

struct Edge{
  int to;
  int cost;
  Edge(){}
  Edge(int a, int b):to(a), cost(b){}
};

const int MAX_N = 100000;

int N;
vector<Edge> G[MAX_N];

int S, T;
int U, V;

void input(){
```

```
  int M;
  scanf("%d%d", &N, &M);
  scanf("%d%d", &S, &T);
  S--; T--;
  scanf("%d%d", &U, &V);
  U--; V--;
  for(int i = 0; i < M; ++i){
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    a--; b--;
    G[a].push_back(Edge(b, c));
    G[b].push_back(Edge(a, c));
  }
}

long long dist[2][MAX_N];

long long dp[MAX_N][4];
long long dist2[MAX_N];
P ps[MAX_N];

priority_queue<P, vector<P>, greater<P> > que;
void dijkstra(int s, long long *res){
  while(!que.empty()) que.pop();
  for(int i = 0; i < N; ++i){
    res[i] = inf;
  }
  res[s] = 0;
  que.push(P(0, s));
  while(!que.empty()){
    P p = que.top();
    que.pop();
    long long c = p.first;
    int v = p.second;
    for(Edge e : G[v]){
      int u = e.to;
      long long nc = c + e.cost;
      if(res[u] <= nc) continue;
      res[u] = nc;
      que.push(P(nc, u));
    }
  }
}

long long get(int v, int id){
  long long res = 0;
  if(id & 1) res += dist[0][v];
  if(id & 2) res += dist[1][v];
  return res;
}

long long solve(){
  for(int i = 0; i < N; ++i){
    for(int j = 0; j < 4; ++j){
      dp[i][j] = inf;
    }
  }
  dijkstra(U, dist[0]);
```

```
  dijkstra(V, dist[1]);
  dijkstra(S, dist2);
  for(int i = 0; i < N; ++i){
    ps[i] = P(dist2[i], i);
  }
  sort(ps, ps + N);
  dp[S][0] = 0;
  dp[S][1] = dist[0][S];
  dp[S][2] = dist[1][S];
  dp[S][3] = dist[0][S] + dist[1][S];
  for(int i = 0; i < N; ++i){
    int v = ps[i].second;
    for(int j = 0; j < G[v].size(); ++j){
      int u = G[v][j].to;
      if(dist2[v] != dist2[u] + G[v][j].cost) continue;
      for(int k = 0; k < 4; ++k){
        for(int l = 0; l < 4; ++l){
          dp[v][k | l] = min(dp[v][k | l], dp[u][k] + get(v, l));
        }
      }
    }
  }
  long long ans = dp[T][3];
  ans = min(ans, dist[0][V]);
  return ans;
}

int main(){
  input();
  long long ans = solve();
  printf("%lld\n", ans);
  return 0;
}
```