گزارش پروژه اول درس مبانی هوش مصنوعی

ایلیا نورانی ۹۹۳۱۰۹۲

۰) کلاس SearchProblems : به طور کلی این کلاس یک طرح کلی از ساختار مساله جست و جو را توصیف می کند (پیاده سازی نمی کند) . در ادامه به توضیح برخی متود های آن می پردازیم:

gerStartState : حالت شروع را برای هر مساله جست و جو بر می گرداند.

isGoalState : اگر استیت ورودی استیت هدف باشد ، true و در غیر این صورت false بر می گرداند.

getSuccessors :لیستی از سه تایی ها بر می گرداند که شامل جانشین (استیت بعدی) ، action(عملی که برای رسیدن به جانشین نیاز است) و stepcost(هزینه رفتن به آن جانشین) می باشد.

getCostOfActions: هزینه کلی دنباله ای از اعمال (action) را بر می گرداند.

متودهای Agent,Direction,Configuration,AgentState,Grid :

Agent : برای پیاده سازی یک عامل بکار می رود که یک اندیس دارد و با توجه به GameState یک حرکت از بین حرکت های Directions انتخاب کند.

Directions : در این کلاس انواع جهت ها و حرکت هایی که عامل می تواند انجام دهد تعریف شده است.

Configuration : به طور کلی مختصات یک کاراکتر را نگه می دارد و متود هایی از جمله مقایسه مختصات و جهت دو کاراکتر را در بر دارد.

AgentState : تمامی استیت ها از جمله speed, configuration,scared و ... را نگه می دارد.

Grid :با استفاده از یک آرایه دو بعدی نقشه بازی را پیاده سازی می کند.

۱) الگوریتم IDS : این الگوریتم در واقع ترکیبی از الگوریتم bfs و dfs است . به طوری که در سطح های مختلف گراف الگوریتم dfs اجرا می شود. همانطور که در کد زیر مشخص است ، تابع IDS در هر سطح(برای این کار از حلقه استفاده شده است) تابع DLS را صدا می زند. این تابع به صورت عمقی تا سطح مشخص شده جست و جو می کند و زمانی که به هدف رسید true و در غیر این صورت false بر می گرداند.

```
bool IDS(src, target, max_depth)
    for limit from 0 to max_depth
        if DLS(src, target, limit) == true
            return true
    return false

bool DLS(src, target, limit)
    if (src == target)
        return true;

    if (limit <= 0)
        return false;

    foreach adjacent i of src
        if DLS(i, target, limit-1)
            return true

    return false
```

۲) الگوریتم BBFS : این الگوریتم به صورت همزمان دو عملیات سرچ را انجام می دهد:

Forward search: از گره اول شروع می کند و تا گره هدف حرکت می کند.

Backward search : از گره هدف شروع می کند و تا گره اول حرکت می کند.

در واقع الگوریتم BBFS از دو زیر گراف تشکیل شده است که یکی مربوط به Forward search و دیگری برای Backward search است . هر گاه این دو زیر گراف هم دیگر را قطع کنند ، الگوریتم به پایان می رسد و مسیر مورد نظر چاپ می شود.

در شبه کد زیر این الگوریتم پیاده سازی شده است. برای این که بتوانیم در مساله جست و جو به دنبال چند هدف باشیم ، باید بر روی تمام گره های هدف ، عملیات Backward search را اجرا کنیم و اگر زیر درختی که در Forward search تشکیل می شود را قطع کردند ، کوتاه ترین مسیر پیدا می شود. پیچیدگی این الگوریتم در برابر الگوریتم bfs بسیار کمتر است . به طوری که اگر ضریب انشعاب b باشد و فاصله هدف تا گره اول d باشد ، o(2 * b^(d/2)) برای BBFS و o(b^d) برای BFS می باشد.

```java
public int biDirSearch(int s, int t)
    {
        Boolean[] s_visited = new Boolean[V];
        Boolean[] t_visited = new Boolean[V];

        int[] s_parent = new int[V];
        int[] t_parent = new int[V];

        Queue<Integer> s_queue = new LinkedList<Integer>();
        Queue<Integer> t_queue = new LinkedList<Integer>();

        int intersectNode = -1;

        for (int i = 0; i < V; i++) {
            s_visited[i] = false;
            t_visited[i] = false;
        }

        s_queue.add(s);
        s_visited[s] = true;

        s_parent[s] = -1;

        t_queue.add(t);
        t_visited[t] = true;

        t_parent[t] = -1;

        while (!s_queue.isEmpty() && !t_queue.isEmpty()) {

            bfs(s_queue, s_visited, s_parent);
            bfs(t_queue, t_visited, t_parent);

            intersectNode = isIntersecting(s_visited, t_visited);

            if (intersectNode != -1) {

                System.out.printf("Path exist between %d and %d\n", s, t);
                System.out.printf("Intersection at: %d\n", intersectNode);
                printPath(s_parent, t_parent, s, t,intersectNode);

                System.exit(0);
            }
        }
        return -1;
    }
```

۳) بله – در الگوریتم UCS هزینه هر گره برابر با هزینه تجمعی می باشد که تا رسیدن به آن گره صرف شده است .

برای پیاده سازی این الگوریتم از یک صف اولویت استفاده می شود که هر بار گره ای که هزینه کمتری (اولویت بیشتر) دارد را گسترش می دهد. حالا اگر هزینه هر گره را بر مبنای جایگاه آن گره در صف تعریف کنیم ، به الگوریتم BFS می رسیم .یعنی هر گره ای که زود تر وارد صف شود اولویت بالاتری داشته برای گسترش داشته و زودتر از صف خارج می شود . اما اگر بر خلاف BFS گره ای که زودتر وارد شده اولویت کمتری داشته باشد و دیرتر از صف خارج شود به الگوریتم DFS می رسیم.

*برای پیاده سازی می توانیم یک متغیر شمارنده تعریف کنیم که با پیمایش هر successor مقدار آن یک عدد افزایش می یابد و آن را به عنوان هزینه گره قرار دهیم و در صف اولویت push کنیم. اگر اولویت را با گره دارای هزینه بالاتر قرار دهیم الگوریتم DFS و اگر با گره دارای هزینه پایین تر قرار دهیم الگوریتم BFS بدست می آید.

۴) الگوریتم dfs بیشترین هزینه را در بین تمامی الگوریتم ها متحمل می شود (۲۹۸ در ۰.۰ ثانیه). بقیه الگوریتم ها هزینه مشابهی دارند (۵۴) با این تفاوت که الگوریتم های bfs وucs در ۰.۰ ثانیه و الگوریتم *A در ۰.۱ ثانیه این هزینه را صرف می کنند.در مورد گره های گسترش داده شده نیز رابطه زیر برقرار است :

A*(535) < dfs(576) < bfs = ucs(682)

۱۶) هیوریستیک به شکل زیر تعریف شده است:

$h =$ فاصله نزدیکترین گوشه تا دورترین گوشه (currentToClosest) + فاصله گره تا نزدیکترین گوشه (closestToFarthest)

هیوریستیک فوق غیر بدبینی و ناسازگار می باشد (فاصله یک عدد بزرگتر یا مساوی صفر است). برای نشان دادن سازگاری باید رابطه زیر را اثبات کنیم:

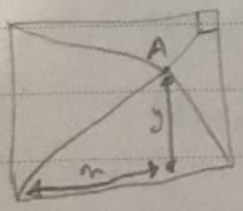
$$h(A) - h(B) \leq cost(A \text{ to } B)$$

اثبات:

$$h(A) = A \text{ To closest} + \text{closest To Farthest}$$

$$h(B) = B \text{ to closest} + \text{closest To Farthest}$$

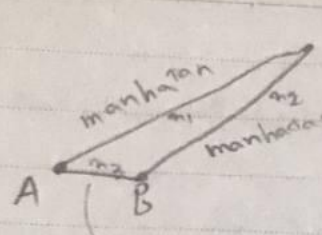
* منظور از فاصله، فاصله منهتن است. با توجه به شکل زیر، مشخص است که دورترین گوشه، گوشه روبه روی نزدیک ترین گوشه است.



در نتیجه:

$$h(A) - h(B) = A \text{ To closest} - B \text{ To closest}$$

در شکل زیر داریم:



در مثلث رو به رو تا حدی $n_2+n_3>n_1$ برقرار است

$n_3 > n_1 - n_2$ ← euclidean ≥ manhattan

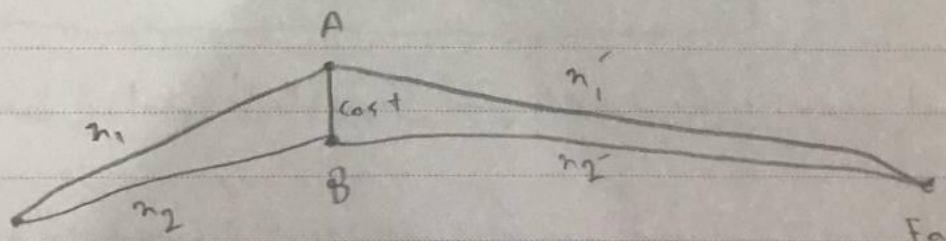
cost A to B

cost A to B $\geq h(A)-h(B)$ ← cost A to B $> n_1 - n_2$

√√

۷) بخش اول، هیوریستیک را مانند قسمت قبل در نظر می‌گیریم.

$h(A)-h(B) \leq cost(A \text{ to } B)$



closest

$n_1' < n_2' + cost \rightarrow n_1' - n_2' < cost$

$n_1 < n_2 + cost \longrightarrow n_1 - n_2 < cost$

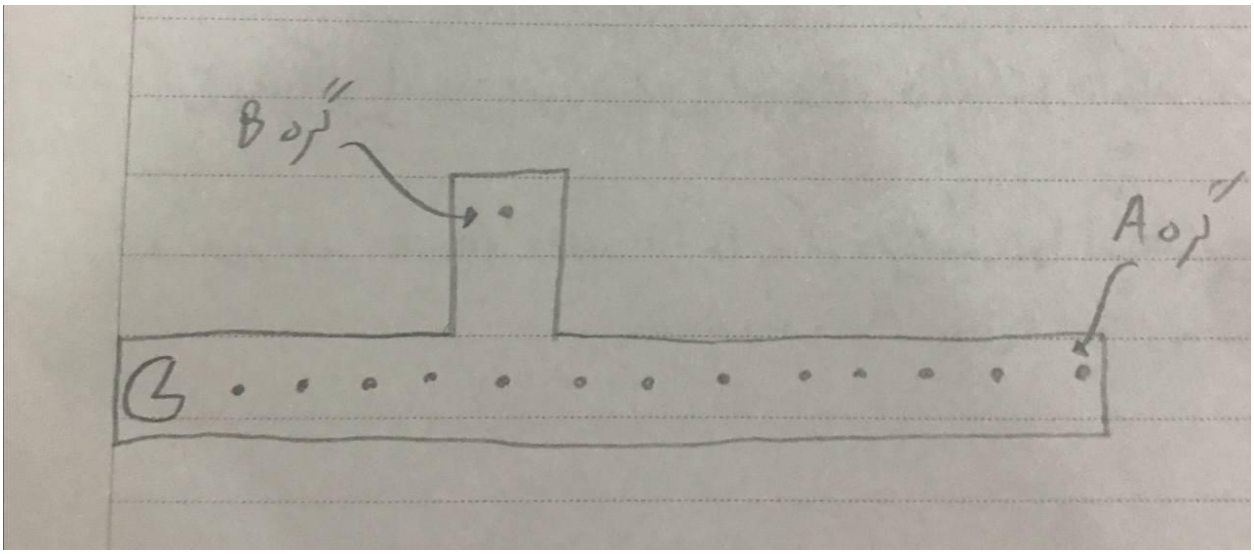
$\rightarrow n_1' + n_1 - n_2' - n_2'' < cost$

$\longrightarrow h(A)-h(B) < cost(A \text{ to } B)$ √√

P4PCO

۷) بخش دوم : هیوریستیک در هر دو قسمت مشابه یکدیگر می باشند با این تفاوت که در قسمت قبل فاصله گره تا هر گوشه محاسبه می شد اما در این قسمت به جای گوشه ، فاصله تا غذا حساب می شود (نزدیک ترین و دور ترین غذا).

۸) در مثال زیر با الگوریتم حریصانه به کوتاه ترین مسیر نمی رسیم (عامل ابتدا به سمت گره A و در نهایت به سمت گره B می رود در صورتی که گره B باید زودتر پیمایش شود).



گزارش کلی کد :

: util.py

متود update :این متود یک ایتم به همراه اولویت آن را به عنوان ورودی می گیرد و در نهایت صف اولویت را آپدیت می کند . اگر این آیتم از قبل داخل صف نبود آن را پوش می کند ، در غیر این صورت با توجه به اولویت جدید و مقایسه آن با اولویت قبلی ، آن را تغییر می دهد (اولویت بالاتر را قرار می دهد).

: Search.py

depthFirstSearch : این متود در واقع الگوریتم dfs را پیاده سازی می کند .

breadthFirstSearch : این متود الگوریتم bfs را پیاده سازی می کند.

uniformCostSearch : این متود الگوریتم ucs را پیاده سازی می کند.

aStarSearch : این متود الگوریتم *A را پیاده سازی می کند.

: searchAgents

manhattanHeuristic : فاصله منهتن دو نقطه را بر می گرداند.

euclideanHeauristic : فاصله اقلیدسی دو نقطه را می دهد.

: cornersProblem

init : یک آرایه به طول ۴ تعریف می کند و مقدار اولیه همه خانه های آن را false می گذارد(یعنی هیچ یک از گوشه ها پیمایش نشده اند) . در اصل این متود برای initialize کردن برخی اجزا می باشد.

Getsuccessor : جانشین را به انتهای لیست successors اضافه می کند با این شرایط که اگر یکی از گوشه ها بود ، مقدار آن گوشه را true می کند که یعنی آن گوشه پیمایش شده است.

Corner heauristic :این متود برای یاده سازی هیوریستیک برای مثله پیدا کردن گوشه می باشد . در قسمت سوال مربوط به این بخش ، کد کامل توضیح داده شده است.

foodSearchProblem : مانند متود Corner heauristic است ، به طور کامل در قسمت سوال مربوط به این بخش توضیح داده شده است.

findPathToClosestDot :

یک مسیر تا نزدیک ترین نقطه را بر میگرداند (برای پیاده سازی از هر الگوریتم سرچی می توان استفاده کرد که در اینجا با الگوریتم dfs پیاده سازی شده است).