گزارش پروژه اول

اشکان شکیبا (9931030)

سوال صفر

کلاس SearchProblem یک کلاس انتزاعی (abstract) است که ساختار کلی یک مسئله جست‌وجو را تعیین می‌کند.

متد getStartState وضعیت آغازین مسئله را بازگردانی می‌کند.

متد isGoalState یک ورودی state می‌گیرد و در صورتی که وضعیت هدف باشد، True بازگردانی می‌کند.

متد getSuccessors نیز یک ورودی state می‌گیرد و لیستی از مجموعه سه‌تایی successor، action و step cost بازگردانی می‌کند که شامل successorهایی بر وضعیت فعلی به همراه action مورد نیاز برای رسیدن به آن و هزینه انجام آن است.

متد getCostOfActions یک ورودی actions می‌گیرد که شامل لیستی از کنش‌هاست و مجموعه هزینه انجام دنباله آنها را بازگردانی می‌کند.

کاربرد کلاس‌های فایل game.py:

Agent: پیاده‌سازی عامل‌ها و بازگردانی کنش‌های آن‌ها

Directions: ذخیره جهت‌های جغرافیایی و جهت‌های نسبی

Configuration: نگه‌داری مختصات موقعیت و جهت حرکت عامل

AgentState: نگه‌داری وضعیت یک عامل، شامل موقعیت، جهت و سرعت حرکت و ...

Grid: نگه‌داری زمین بازی در قالب یک آرایه دو بعدی

سوال یک

در الگوریتم IDS، بخشی از گراف که تا قبل از عمق تعیین‌شده‌ای قرار دارد انتخاب شده و در آن بخش به جست‌وجو با DFS می‌پردازیم. به بیان دیگر الگوریتم DFS از عمق تعیین‌شده پایین‌تر نمی‌رود و در صورتی که در محدوده انتخاب شده به هدف نرسیم، مقدار عمق تعیین‌شده به مرور افزایش می‌یابد.

بنابراین می‌توان گفت برای تغییر DFS به IDS باید در چند مرحله و تا زمان رسیدن به هدف، عمق مشخصی تعیین کرد که الگوریتم DFS در هر مرحله از عمق مشخص‌شده در آن پایین‌تر نرود.

شبه کد الگوریتم IDS:

```
function IDS(root) is
    for depth from 0 to ∞ do
        found, remaining ← DLS(root, depth)
        if found ≠ null then
            return found
        else if not remaining then
            return null
```

```
function DLS(node, depth) is
    if depth = 0 then
        if node is a goal then
            return (node, true)
        else
            return (null, true) (Not found, but may have children)

    else if depth > 0 then
        any_remaining ← false
        foreach child of node do
            found, remaining ← DLS(child, depth−1)
            if found ≠ null then
                return (found, true)
            if remaining then
                any_remaining ← true
                 (At least one node found at depth, let IDS deepen)
        return (null, any_remaining)
```

سوال دو

تفاوت الگوریتم BBFS در این است که جست‌وجویی مانند BFS را از دو سو شروع می‌کنیم، یکی به شکل پایین‌رونده از راس آغازین و یکی به شکل بالارونده از راس هدف. فرایند زمانی به پایان می‌رسد که دو جست‌وجو در یک نقطه به همدیگر برسند.

شبه کد الگوریتم BBFS:

```
BIDIRECTIONAL_SEARCH
 1   Q_I.Insert(x_I) and mark x_I as visited
 2   Q_G.Insert(x_G) and mark x_G as visited
 3   while Q_I not empty and Q_G not empty do
 4       if Q_I not empty
 5           x ← Q_I.GetFirst()
 6           if x = x_G or x ∈ Q_G
 7               return SUCCESS
 8           forall u ∈ U(x)
 9               x' ← f(x, u)
10               if x' not visited
11                   Mark x' as visited
12                   Q_I.Insert(x')
13               else
14                   Resolve duplicate x'
15       if Q_G not empty
16           x' ← Q_G.GetFirst()
17           if x' = x_I or x' ∈ Q_I
18               return SUCCESS
19           forall u^-1 ∈ U^-1(x')
20               x ← f^-1(x', u^-1)
21               if x not visited
22                   Mark x as visited
23                   Q_G.Insert(x)
24               else
25                   Resolve duplicate x
26   return FAILURE
```

در صورتی که بیش از یک هدف داشته باشیم، می‌توان به شکل همزمان جست‌وجوی بالارونده از آنها را پیش برد تا یکی از آنها در نقطه‌ای به جست‌وجوی پایین‌رونده از مبدا برسد.

سوال سه

بله، ممکن است.

برای تبدیل UCS به BFS، با توجه به پیش‌روی لایه‌ای BFS، باید کمترین هزینه مربوط به لایه اول باشد و با حرکت به سمت لایه‌های بعدی هزینه بیشتر شود.

برای تبدیل UCS به DFS، هزینه‌ها مشابه BFS اما به شکل عمقی و نه لایه‌ای افزایش می‌یابند.

واضح است که در هر دوی این حالت، باید همچنان از صف اولویت استفاده کنیم.

سوال چهار

در DFS پاسخ به دست آمده اصلا بهینه نیست و مسیری نسبتا طولانی‌ست.

در BFS به پاسخی بهینه می‌رسیم اما زمان محاسبه الگوریتم نسبتا بالاست. UCS نیز مشابه BFS عمل می‌کند.

در A* پاسخ بهینه است و همچنین با زمان کمتر و بررسی خانه‌های کمتری نسبت به الگوریتم‌های قبلی، به پاسخ بهینه دست می‌یابد. که این تعداد نیز وابسته به تابع هیوریستیک ماست و برای هیوریستیک منهتن نسبت به اقلیدسی، خانه‌های کمتری را بررسی می‌کند.

سوال شش

در هیوریستیک پیاده‌سازی شده، هر بار فاصله منهتن تا گوشه‌های باقی‌مانده محاسبه می‌شود و عامل به سمت گوشه نزدیک‌تر حرکت می‌کند.

از آن جایی که در هر بخش از مسیر، در بهترین حالت که هیچ مانعی نباشد، با توجه به نحوه حرکت پک‌من، هزینه برابر فاصله منهتن می‌شود و در صورت وجود موانع، افزایش می‌یابد، می‌توان نتیجه گرفت که هیوریستیک قابل قبول و سازگار است.

سوال هفت

- این هیوریستیک بر اساس محاسبه فاصله دورترین غذا و بازگردانی آن کار می‌کند. دلیل سازگار بودن آن استفاده از تابع mazeDistance است که در هر بخشی از مسیر، مقداری نابزرگ‌تر از هزینه برمی‌گرداند.
- در هیوریستیک سوال قبل نزدیک‌ترین مکان را پیدا و از آن شروع می‌کردیم، اما در این هیوریستیک الزامی بر این کار نیست.

سوال هشت

واضح است که نمی‌توان با رفتن به نزدیک‌ترین خانه‌ای که غذا در آن هست، مطمئن بود که مسیری بهینه را می‌پیماییم.

برای مثال در تصویر زیر که مربوط به حل bigSearch با ClosestDotSearchAgent است، می‌بینیم که یک نقطه در سمت راست ماز باقی مانده که در نهایت پک‌من باید برای خوردن آن برگردد و به وضوح مسیر انتخاب شده با روش حریصانه، بهینه نیست.