۰)

کاربرد کلاس و متودهای SearchProblem :

این کلاس یک کلاس ابسترکت است، به این معنی که متودهای آن پیاده‌سازی نشده و هر متود در کلاس‌هایی که از این کلاس ارث‌بری کرده‌اند به صورت مجزا پیاده‌سازی شده است. کاربرد این کلاس در مشخص کردن ساختار یک مسئله‌ی جست‌وجو است. کاربرد متودهای آن نیز به ترتیب برای مشخص کردن وضعیت شروع مسئله، مشخص کردن این که آیا در مسأله به حالت هدف رسیده‌ایم یا خیر، گرفتن successor به صورت یک لیست سه‌تایی متشکل از حالت بعدی برای حالت کنونی، actionی که برای رسیدن به آن حالت باید انجام دهیم، و همچنین هزینه‌ی این action، و متود آخر نیز برای محاسبه‌ی هزینه ی کل دنباله‌ای از حرکت‌های مجاز است.

کاربرد کلاس‌های game.py :

agent : یک متود دارد که یک آبجکت از کلاس Gamestate را دریافت کرده و یک عمل که یکی از موارد شمال، جنوب، شرق، غریب و توقف است را خروجی می‌دهد.
Directions : این کلاس صرفاً دارای مقادیری استاتیک است که از آن‌ها برای تعیین جهت حرکت‌های عوامل در طول بازی استفاده می‌شود.
Configurations : این کلاس مختصات یک کاراکتر، و همچنین جهت حرکت آن را در خود نگه می‌دارد.
AgentState : در خود وضعیت یک عامل که شامل چیزهایی مانند مختصات، جهت حرکت، سرعت، زمان scared بودن و ... را نگه می‌دارد.
Grid : آرایه‌ای دو بعدی از اشیاء که با لیستی از لیست‌ها پیاده‌سازی شده است و در خود عوامل بازی را نگه می‌دارد.
Actions : دارای متودهایی استاتیک برای تعیین حرکت‌های عوامل بازی و تبدیل بردار به این حرکت‌ها و برعکس است. از این کلاس به همراه کلاس Directions برای حرکت عوامل بازی استفاده می‌شود.

۱)

بخش اول:

پیچیدگی زمانی این الگوریتم برابر با است که $O(b^m)$ در آن branching factor و m در آن ارتفاع درخت است، زیرا در این الگوریتم در صورت این که حالت مورد نظر آخرین حالتی باشد که از پشته بیرون می‌آید، باید کل گره‌های درخت یک بار بررسی شوند که تعدادشان از این order است.
همچنین پیچیدگی فضایی این الگوریتم برابر با bm است. زیرا در هر مرحله باید با پیش رفتن در عمق درخت، یک گره و گره‌های خواهر و برادر آن را در پشته نگه داریم، و این کار را برای m گره که ارتفاع درخت است انجام می‌دهیم. پس تعداد آن‌ها برابر با این مقدار می‌شود.

خیر، استفاده از آن منطقی و بهینه نیست. زیرا در هر مرحله توجهی به هزینه‌ی صرف شده و نزدیکی به هدف نداشته و فقط عمیق‌ترین گره را انتخاب می‌کند. این جست‌وجو به‌خصوص در مواقعی که اولویت با بررسی شاخه‌هایی از سمت خاصی از درخت باشد و حالات هدف در سمتی دیگر و در عمق زیادی باشند، حالت بهینه نیست.

بخش دوم:

```
IDS(root, goal) :
    for I = 0 to depth_of_tree :
            if DFS(root, goal, I) :
                    return True
        return False
```

این الگوریتم درواقع DFS را هر بار با محدودیت تعداد لایه‌ها اجرا می‌کند. مثالی که می‌توان برای این که عمل‌کرد IDS بدتر از DFS زد، این است که در الگوریتم DFS از سمت چپ درخت شروع کنیم و در عمق‌های مساوی نیز اولویت با گره‌های سمت چپ باشد، و حالت هدف نیز در عمق زیادی و در سمت چپ درخت باشد. حال با استفاده از الگوریتم DFS با صرف زمانی نسبتاً کم می‌توان به این گره رسید، در حالیکه در الگوریتم دوم باید هر بار DFS بر روی تمامی لایه‌ها با عمق کم‌تر از گره‌ی هدف اجرا شود تا سرانجام به گره‌ی هدف برسیم، و مشخصاً آین باعث می‌شود که نسبت به الگوریتم DFS عمل‌کرد بدتری داشته باشد.

۲)

بخش اول:

بله. در این مسأله نیز کد با استفاده از الگوریتم BFS دنباله‌ای از حرکات را که منجر به حل این مسأله می‌شود تولید کرده و مساله را حل می‌کند..

بخش دوم:

الگوریتم BBFS که اختصاریافته‌ی Bidirectional BFS است، در واقع از همان الگوریتم BFS استفاده می‌کند، ولی این کار را به صورت هم‌زمان و با شروع از دو راس مختلف انجام می‌دهد، به این معنی که این الگوریتم به صورت همزمان و یک بار برای راس ریشه و یک بار برای راس هدف اجرا می‌شود، و هنگامی خاتمه می‌یابد که گره‌های بسط داده شده برای این دو الگوریتم به هم برسند(تداخل پیدا کنند). این الگوریتم بهینه‌تر از الگوریتم BFS است، زیرا درواقع گراف مسأله را به دو زیرگراف تقسیم می‌کند، و این باعث می‌شود که راه‌حل مسأله در صورت وجود در زمان کم‌تری کشف شود. شبه‌کد آن به صورت زیر است:

```
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
```

بخش سوم:

پیچیدگی زمانی و فضایی الگوریتم BFS هر دو از مرتبه‌ی $O(b^s)$ است که در آن b همان branching factor و s لایه‌ای از درخت است که هدف در آن قرار گرفته است. زیرا در واقع برای پیدا کردن حالت هدف به صورت لایه لایه به پایین می‌رویم که در نهایت تعداد گره‌های بررسی‌شده از مرتبه‌ی تعداد گره‌های لایه‌ی آخر است، و همچنین هر بار نیز باید گره‌های لایه‌ی آخر را ذخیره کرده باشیم در نتیجه مرتبه‌ی آن باز همین اندازه می‌شود. در صورتی که حالات هدف در عمق کمی از درخت باشند، الگوریتم BFS عمل‌کرد بهتری دارد، و در غیر این صورت استفاده از الگوریتم DFS می‌تواند بهتر باشد. البته در این‌جا تعیین اولویت در هنگام انتخاب بین گره‌های با اولویت یکسان نیز مورد اثرگذاری می‌تواند باشد و در نتیجه این که گره‌ی هدف در کدام سمت از درخت باشد نیز می‌تواند اثرگذار باشد.

۳)

بخش اول:

برای رسیدن از الگوریتم UCS به الگوریتم BFS، می‌توانیم تابع هزینه را به گونه‌ای تعریف کنیم که هزینه برای همه‌ی یال‌های گراف برابر باشد. برای مثال می‌توانیم هزینه‌ی همه‌ی یال‌ها را برابر با یک قرار بدهیم. در این صورت با توجه به این که الگوریتم UCS در هر مرحله گره‌ای را انتخاب می‌کند که مسیر از ریشه تا آن گره کمترین بوده، این الگوریتم گره‌ها ها به صورت لایه لایه انتخاب می‌کند که رفتاری شبیه به BFS است. در کد باید هنگامی که گره‌ها را به صف وارد می‌کنیم، به جای هزینه‌ی هر گره ۱ بگذاریم.
برای رسید از الگوریتم UCS به الگوریتم DFS، باید تابع هزینه را به شکلی تعریف کنیم که هر چه به سمت عمق می‌رویم، هزینه کم‌تر شده و در نتیجه گره‌های عمیق‌تر زودتر گسترش یابند. می‌توان فهمید که برای این هدف، می‌توان به راحتی هزینه‌ی یال‌ها را برابر با یک عدد منفی یکسان بگذاریم. برای مثال وزن همه‌ی یال‌ها را برابر با منفی یک قرار بدهیم. در این صورت با گسترش یافتن گره‌ها، هر گره در عمق بیش‌تر هزینه‌ای کمتر داشته و زودتر گسترش می‌یابد. در کد باید هنگامی که گره‌ها را به صف وارد می‌کنیم، به جای هزینه‌ی هر گره منفی یک بگذاریم.

بخش دوم:

این الگوریتم علاوه بر کامل بودن که دو الگوریتم ناآگاهانه‌ی دیگر هم آن را داشتند، مانند BFS بهینه است، پس می‌توان همین را یک برتری بزرگ این الگوریتم بر DFS دانست. همچنین UCS برخلاف دو الگوریتم دیگر بر روی گراف‌هایی با یال‌هایی با وزن‌های متفاوت نیز به درستی عمل می‌کند. اما از معایب این الگوریتم می‌توان به این مورد اشاره کرد که جست‌وجو را در تمام جهات انجام می‌دهد زیرا اطلاعی از مکان گره‌ی هدف و میزان نزدیکی به آن ندارد. این ایراد در الگوریتم بعدی مطرح شده در پروژه و با استفاده از تابع huristic حل می‌شود.

۴)

بخش اول:

از قبل می‌دانیم که در صورت پیاده‌سازی درست توابع الگوریتم‌ها، سه الگوریتم BFS, UCS و A* باید در انتهای جست‌وجو مسیر بهینه را به ما بدهند. با اجرا کردن این سه الگوریتم، می‌بینیم که در عمل نیز این اتفاق می‌افتد و هر سه مسیرهایی با هزینه‌ی ۵۴ را به ما تحویل می‌دهند. همچنین مطابق با پیش‌بینی، به ترتیب الگوریتم A* UCS و BFS عمل‌کرد بهتری نسبت به الگوریتم بعدی خود در این لیست دارند. در واقع با اجرای این الگوریتم‌ها، به ترتیب این الگوریتم‌ها ۵۳۵، ۶۸۲ و ۶۸۲ گره را هنگام محاسبه‌ی مسیر بهینه بسط می‌دهند. نکته‌ی قابل توجه این است که برای این مثال خاص، دو الگوریتم آخر به ما دقیقاً یک راه‌حل و با یک تعداد بسط دادن تحویل داده‌اند

. همچنین از قبل می‌دانیم که DFS الزاماً مسیر بهینه را به ما نمی‌دهد، و هنگام اجرای آن نیز مشخص است که مسیری که این الگوریتم به ما می‌دهد به هیچ‌وجه بهینه نیست. در واقع هنگام اجرای این الگوریتم، مسیر یافت‌شده دارای هزینه‌ی کلی ۲۹۸ است و همچنین برای این کار ۵۷۶ گره را بسط می‌دهد.

بخش دوم:

ایده‌ی اصلی الگوریتم Dijkstra این است که همه‌ی مسیرها را از نقطه‌ی شروع تا نقطه‌ی هدف explore می‌کند. این الگوریتم این کار را با نگه داشتن یک لیست از گره‌های مشاهده‌نشده و به صورت تدریجی اضافه کردن گره‌هایی با کمترین هزینه انجام می‌دهد. در واقع ایده‌ی این الگوریتم تا حد زیادی شبیه به الگوریتم دیگر است، به جز این که ما در الگوریتم A* با داشتن یک heuristic که فاصله‌ی هر گره را تا مقصد تخمین می‌زند، درواقع درخت جست‌وجو را هرس می‌کنیم و باعث می‌شود که گره‌ها و مسیرهای کمتری امتحان شوند. این کار با استفاده از یک صف اولویت انجام می‌شود که در هر مرحله هم مسافت پیموده‌شده تا رسیدن به آن گره و هم مسافت تخمینی مانده تا گره‌ی هدف را به عنوان هزینه در نظر می‌گیرد. پس در کل می‌توان الگوریتم دوم را بهینه‌تر و مناسب‌تر از الگوریتم اول یافت.

۶)

ما در این‌جا هیوریستیک خود را برابر با (حداکثر فاصله‌ی منهتن از موقعیت فعلی پک‌من تا گوشه‌های ویزیت‌نشده) در نظر گرفته‌ایم. قابل قبول بودن این هیوریستیک به سادگی قابل اثبات است. فاصله‌ی منهتن در واقع کوتاه‌ترین فاصله‌ای است که پک‌من می‌تواند از نقطه‌ای به نقطه‌ی دیگر برود، و این بدون در نظر گرفتن موانع سر راه آن است. حال اگر موانع در نظر گرفته شوند، این فاصله مساوی با بیش‌تر از فاصله‌ی منتهتن می‌شود.
همچنین برای سازگار بودن، می‌دانیم که نباید در طول مسیر حرکت پک‌من، f(n) کاهش یابد. حال با در نظر گرفتن این که با استفاده از الگوریتم هر بار پکمن به سمت گوشه‌ای با هزینه‌ی کم‌تر می‌رود، و با توجه به این که در طول این مسیر مسافت پیموده‌شده و همچنین تابع هیوریستیک که درواقع حداکثر فاصله تا گوشه‌ها است افزایش می‌یابند، پس در کل تابع هزینه نیز افزایش می‌یابد و سازگاری این هیوریستیک اثبات می‌شود.

۷)

بخش اول:

ما در این‌جا هیوریستیک را برابر با (حداکثر mazeDistance موقعیت فعلی پک‌من تا نقاط خورده‌نشده) در نظر می‌گیریم. دلیل سازگاری آن به کارگیری تابع mazeDistance است که در هر بخشی از مسیر با استفاده از BFS فاصله‌ی دو نقطه را برمی‌گرداند که به دلیل یکسان بودن هزینه‌ها بهینه است. و سپس به همان دلیل سؤال قبلی، می‌توان سازگاری این هیوریستیک را اثبات کرد.

بخش دوم:

یکی از تفاوت‌ها در استفاده از توابع برای تخمین فاصله در طول مشخص کردن هیوریستیک است. در هیوریستک قبلی از تابع manhattanDistance و در این هیوریستیک از mazeDistance که خود از BFS استفاده می‌کند استفاده کرده‌ایم. همچنین در قبلی ما هنگام اجرای تابع هیوریستیک برای اولین بار برای تمام خانه‌های عیردیوار و همچنین تمام غذاها در ابتدا فاصله‌ها را پیدا کرده و در ساختمان داده‌ی مپ ذخیره می‌کنیم و در دفعات بعد نیز چون موقعیت پک‌من و همچنین غذاها نمی‌تواند چیزی غیر از موارد ذخیره‌شده باشد با استفاده از دسترسی سریعی که دیکشنری ذخیره‌شده به ما می‌دهد از همان مقادیر استفاده می‌کنیم، در حالیکه در قبلی بار هر بار فراخوانی تابع ما فاصله‌ی موقعیت پک‌من تا گوشه‌ها را حساب می‌کنیم و سپس از بین آن‌ها ماکسیمم می‌گیریم.

بخش سوم:

زیرا این الگوریتم بر خلاف سه الگوریتم BFS, DFS و UCS، می‌تواند با استفاده از تابع هیوریستیک تعریف شده برای آن میزان نزدیک بودن به هدف در هر مرحله را نیز بسنجد، و این سبب می‌شود که درخت جست‌وجو هرس بشود و سرعت رسیدن به جواب بهینه بیش‌تر بشود.

۸)

واضح است که با استفاده از روش حریصانه نمی‌توان الزاماً به کوتاه‌ترین مسیر رسید، زیرا این الگوریتم در هر مرحله نزدیک‌ترین نقطه را با توجه به موقعیت فعلی عامل پیدا می‌کند و دیدی از حرکات بعدی که ممکن است در کل منجر به طولانی شدن مسیر بشوند ندارد. مثال آن در پیاده‌سازی تابع با استفاده از الگوریتم IDS است که همان‌طور که مشخص است، عامل پس از خوردن اکثر نقاط یک سمت از صفحه، دو نقطه را رها کرده و در آخر باید بخش زیادی از نقشه را بازگردد تا آن‌ها را بخورد: