گزارش تمرین
کامپیوتری اول

ترم پائیز 1403

درس هوش مصنوعی

امیرمرتضی رضائی - 810003004

مدل‌سازی مسئله:

Environment:

ماتریس $n \times n$ که شامل وضعیت هرلامپ می‌باشد.

Action:

عمل تغییر وضعیت هرلامپ که علاوه بر خود، منجر به تغییر وضعیت ۴ لامپ همسایه با آن نیز می‌گردد.

Initial State:

ماتریسی که در ابتدا به عنوان ورودی به agent ما داده می شود.

Goal State:

صفر شدن تمام درایه‌های ماتریس اولیه

جستجوی ناآگاهانه BFS:

الگوریتم BFS برای جستجوی مقداری خاص در یک درخت یا گراف استفاده می‌شود. روال جستجوی الگوریتم BFS از گره ریشه درخت یا گراف آغاز می‌شود. در هر سطح از درخت یا گراف، تمامی گره‌ها مورد بررسی قرار می‌گیرند و سپس روند جستجو در سطح بعدی این ساختار داده‌ها ادامه پیدا می‌کند. با کمک این الگوریتم می‌توان بدون گیر افتادن در یک حلقه بی‌پایان، هر گره را بررسی کرد.

الگوریتم جستجوی اول سطح از ساختار داده صف (Queue) برای پیمایش گراف یا درخت استفاده می‌کند. یکی از اصول ساختار داده صف، اصل اولین ورودی - اولین خروجی (FIFO) است. الگوریتم BFS با استفاده از چنین اصلی، در هر گامی که گره جدیدی را در گراف یا درخت ملاحظه می‌کند، گره‌های مجاور (گره‌های فرزند) آن را در صف قرار می‌دهد و سپس گره‌های موجود در صف را با اصل FIFO برای یافتن پاسخ، بررسی می‌کند.

برای پیاده‌سازی این الگوریتم، مطابق شبه‌کد زیر عمل شده‌است:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

در واقع، در یک حلقه while ابتدا به آخرین state ای که به آن دست‌یافته‌ایم تغییر وضعیت داده و بررسی می‌کنیم که goal state محقق شده‌باشد. در این‌صورت خروجی را بازگردانده و از برنامه خارج می‌شویم. ولی در غیراین صورت، تمام فرزاندان state کنونی را تولید کرده و در صورتی که قبلا بررسی نشده باشند، آن‌ها را در صف قرار می‌دهیم. (بررسی ملاقات نشدن stateها توسط لیستی به‌نام visited انجام می‌پذیرد. درنهایت نیز عنصر اول صف را حذف می‌کنیم. این روند در بدترین حالت تا خالی شدن لیست ادامه خواهد یافت.(حالتی که هیچ پاسخی یافت نشود) $O(b^d)$).

توجه گردد که باتوجه به صورت مسئله، این الگوریتم برای nهای بزرگتر و مساوی ۵ محدود شده‌است.

(نتایج تست‌ها در پایان گزارش بررسی شده‌اند.)

جستجوی ناآگاهانه IDS:

یک استراتژی جستجوی فضای حالت است که در آن یک جستجوی عمق محدود، بارها و بارها اجرا می‌شود که با هر تکرار حد عمق را افزایش می‌دهد. IDS، مشابه جستجوی اول سطح است با این تفاوت که حافظه‌ی کمتری را اشغال می‌کند؛ در هر تکرار، گره‌هایی را که در درخت جستجو در همان سطح از جستجوی عمق اول هستند را می‌بیند، اما مرتبه‌ی تجمعی برای هر گره که اولین بار دیده می‌شود اول سطح است.

در واقع در این روش، مزایای روش‌های DFS و BFS تجمیع شده‌اند.

برای پیاده‌سازی این الگوریتم، مطابق شبه‌کد زیر عمل شده‌است:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

که در آن، شبه‌کد تابع DLS مطابق زیر است:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? ← false
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit − 1)
            if result = cutoff then cutoff_occurred? ← true
            else if result ≠ failure then return result
        if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

برای پیاده‌سازی تابع DLS، در ابتدا رسیدن به goal state و سپس شرط مثبت بودن depth بررسی می‌گردد. سپس تمام فرزندان state کنونی تولید شده و در صورتی که قبلا بررسی نشده باشند، وارد صف می‌شوند و در هرمرحله، همین تابع به‌صورت بازگشتی و با depth یک واحد کمتر بازخوانی می‌شود. توجه شود که صف در این الگوریتم به‌صورت LIFO عمل می‌کند.

حال در تابع اصلی IDS، در یک حلقه‌ی بی‌نهایت، با شروع از عمق ۰ پیش رفته و در هرمرحله، عمق را ۱ واحد افزایش می‌دهیم.

برای شمارش تعداد stateهای بررسی‌شده، چون تابعی دیگر را فراخوانی می‌کنیم، از اشاره‌گرها استفاده شده‌است. همچنین اردر زمانی این الگوریتم از مرتبه‌ی $O(b^d)$ می‌باشد.

توجه گردد که باتوجه به صورت مسئله، این الگوریتم برای nهای بزرگتر و مساوی ۵ محدود شده‌است.

(نتایج تست‌ها در پایان گزارش بررسی شده‌اند.)

جستجوی آگاهانه A*:

در این روش، برای هر state که visit می‌شود، یک مقدار f را اختصاص می‌دهیم که f مجموع هزینه‌ی رسیدن به آن state و هزینه‌ی تخمینی ما از آن state به goal state می‌باشد.
در این الگوریتم بدین روش عمل می‌کنیم که در هر مرحله، stateای را از fringe انتخاب می‌کنیم که دارای کمترین مقدار f باشد.
در واقع، در یک حلقه while ابتدا به کم‌هزینه ترین stateای که به آن دست‌یافته‌ایم تغییر وضعیت داده و بررسی می‌کنیم که goal state محقق شده‌باشد. در این‌صورت خروجی را بازگردانده و از برنامه خارج می‌شویم. ولی در غیراین صورت، تمام فرزاندان state کنونی را به همراه مقدار f متناظر آن ها تولید کرده و در صورتی که قبلا بررسی نشده باشند، آن‌ها را در صف قرار می‌دهیم. توجه شود که تفاوت مجموعه visited در این‌جا با قسمت‌های قبلی، آن‌ست که در این قسمت، علاوه بر board، مقدار f متناظر با آن نیز ذخیره می‌گردد؛ در واقع ممکن است چند بار یک board یکسان را بررسی کنیم که در هر بار هزینه‌ای متفاوت داشته است.

توابع heuristic :

۱- ساده‌ترین روش برای تخمین هزینه رسیدن به goal آنست که فرض کنیم اعمال تغییر وضعیت در هرلامپ، باعث تغییر وضعیت لامپ‌های دیگر نمی‌شود. پس هزینه هر state برابر خواهد بود با تعداد ۱ های موجود در ماتریس هر state.

۲- در ماتریس‌های با ابعاد بیشتر، اصولا درایه‌هایی که مقدار ۱ دارند، احتمال آن‌که فاصله‌شان از هم بیشتر باشد، بیشتر است. بنابراین روشی دیگر برای تخمین آن است که مربع تعداد ۱ ها را برگردانیم! همانطور که گفته شد، انتظار می‌رود این روش در ماتریس‌های با ابعاد بیشتر، بهتر از روش قبلی عمل کند.

۳- در روش بعدی، همسایه‌های لامپ‌هایی که روشن هستند را بررسی می‌کنیم، و اگر هریک از همسایه‌ها خاموش بودند، به هزینه‌ی تخمینی ۱ واحد اضافه می‌کنیم؛ چرا که خاموش کردن هر لامپ روشن، منجر به روشن شدن این تعداد لامپ خاموش خواهد شد.

پاسخ به سوالات:

۱- در قسمت مدل‌سازی پاسخ داده شد.

۲- در میان الگوریتم‌های ناآگاهانه، الگوریتم IDS بهینه‌تر عمل می‌کند، چرا که معمولا space complexity برای IDS ، $O(bd)$ ست، در حالی که برای BFS، $O(b^d)$ می‌باشد!
همچنین می‌دانیم در صورتی که از heuristic که consistent باشد، استفاده کنیم، الگوریتم A* جواب بهینه را تولید می‌کند. همانطور که نتایج نشان می‌دهند، هنگامی که ابعاد مسئله بزرگ‌تر می‌شوند، الگوریتم‌های ناآگاهانه تقریبا دیگر قادر به حل مسئله نخواهند بود.

۳- در heuristic اول، چون هزینه پیش‌بینی شده از هزینه‌ی اصلی کمتر است، یعنی $h(n) \leq h^*(n)$، این تابع admissible است. همچنین می‌توان نتیجه گرفت که $h(n) \leq c(n, a, n') + h(n')$. بنابراین این تابع consistent نیز می باشد.
در مورد تابع دوم نمی‌توان اظهار نظر کرد. ولی تابع سوم، هردو خاصیت را به همان دلایل بیان شده در بالا دارد.

۴- در نتایج بدست‌آمده، همانطور که انتظار داشتیم، الگوریتم‌های آگاهانه عملکرد بسیار بهتری داشتند. در میان الگوریتم‌های A* نیز در ماتریس‌های 3×3، heuristic سوم بهتر از تابع اول عمل می‌کند و تابع دوم عملکرد چندان مناسبی ندارد. اما با افزایش ابعاد ماتریس ورودی، تابع دوم به طرز چشمگیری عملکرد بهتری دارد. (البته جز در یکی از تست‌ها)
در مورد جستجوی A* وزن‌دار، می‌دانیم زمانی بهینه است که ضریب آن بین 0 و 2 باشد. به‌همین دلیل 2 مقدار 1.3 و 1.6 انتخاب شدند. به‌ازای heuristic 1، عملکرد با ضریب 1.6 بهتر است ولی برای توابع دوم و سوم، عملکرد ضریب 1.3 بهینه‌تر می‌باشد.
همچنین برای توابع دوم وسوم، استفاده از A* وزن‌دار، باعث کاهش کیفیت عملکرد برنامه می‌گردد.

۵- نتایج انجام تست‌ها در فایل خروجی قابل مشاهده می‌باشد.