دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

گزارش پروژه اول

مبانی و کاربردهای هوش مصنوعی

نگارش

آرش حاجی صفی - ۹۶۳۱۰۱۹

دی ۱۳۹۹

فرموله‌سازی مسئله:

برای توضیح نحوه فرموله‌سازی مسئله، ابتدا باید کلاس‌هایی که پیاده‌سازی نموده‌ام را شرح دهم. تمامی کلاس‌ها و متدهایی که مربوط به خود مسئله هست (فارغ از نحوه پیاده‌سازی الگوریتم جستجو)، در فایل data_structure.py قرار داده شده اند و در مابقی فایل‌ها import شده اند. در ادامه به توضیح این موارد که برای فرموله‌سازی مسئله به کار رفته‌اند می‌پردازم:

کلاس Card:

برای هر کارت در مسئله اصلی، یک شیء از این نوع ساخته می‌شود. هر شیء این کلاس دارای فیلدهای number و color می‌باشد که به ترتیب شماره و رنگ یک کارت را مشخص می‌کند.

کلاس State:

هر شیء این کلاس یک وضعیت ممکن در بازی را نشان می‌دهد. این کلاس دارای فیلد rows است که یک آرایه دو بعدی است و هر ردیف آن، لیستی از کارت‌های (اشیاء از جنس کلاس Card) موجود در آن ردیف در بازی اصلی است.

کلاس Node:

هر شیء این کلاس معدل یک گره در طول اجرای الگوریتم می‌باشد. این کلاس دارای فیلدهای state (وضعیت بازی در گره موجود را نشان می‌دهد)، parent (مشخص می‌کند کدام گره این node را ایجاد کرده)، action (مشخص می‌کند در نهایت برای رسیدن به گره هدف، برای این گره کدام عملیات انجام شده تا به گره بعد در مسیر مورد نظر تا هدف برسیم)، par_action (مشخص می‌کند والد این گره با کدام عملیات این node را ایجاد کرده)، و cost (هزینه را که در این مسئله عمق گره فعلی است، مشخص می‌کند) می‌شود.

کلاس Node_H:

دقیقاً از همان کلاس Node ارث بری می‌کند و فیلدهای heuristic (مقدار هیوریستیک برای گره را مشخص می‌کند) و f (همان تابع f = heuristic + cost است) را به آن اضافه می‌کند. هدف از ایجاد این کلاس صرفاً برای سوال سوم می‌باشد.

همچنین این کلاس شامل متد calculate_heuristic() هم می‌شود که کار محاسبه هیوریستیک برای گره از این نوع که آنرا صدا بزند می‌شود و نحوه محاسبه آنرا در توضیحات سوال سوم ارائه می‌دهم.

کلاس Action:

معادل یک action برای جابجایی از یک state به state دیگر است. شامل دو فیلد first_row و second_row می‌شود و مشخص می‌کند که عملیات مورد نظر به این صورت است که آخرین کارت از ردیف first_row را در بازی برداریم و آنرا در انتهای ردیف second_row قرار دهیم.

متدها:

متد get_all_actions(node): شیء از جنس Node را ورودی می‌گیرد و یک لیست از تمام اشیاء Action ممکن مطابق با چیزی که در تعریف پروژه گفته شده بر می‌گرداند.

متد init_game(file_name): رشته ای از نام فایل مورد نظر که تعریف حالت ابتدایی بازی در آن مطابق فرمت ورودی صورت پروژه قرار دارد را می‌گیرد و خط اول را می‌خواند و تعداد ردیف‌ها، تعداد رنگ‌ها، و تعداد کارت‌های موجود از هر رنگ را استخراج می‌کند و در سه متغیر global به نام‌های rows_num, colors_num, cards_num ذخیره می‌کند؛ سپس ادامه فایل را خط به خط می‌خواند و هر ورودی موجود در هر ردیف را تبدیل به یک شیء از جنس Card می‌کند و نهایتاً یک شیء از جنس State بر می‌گرداند که وضعیت اولیه بازی را مشخص می‌کند.

متد goal_test(state): یک شیء از جنس State را می‌گیرد و مطابق با تعریف پروژه، آزمون هدف را روی آن اجرا می‌کند و نسبت به اینکه این حالت حالت هدف است یا نه، True یا False برمی‌گرداند.

متد get_solution(node): یک Node را ورودی می‌گیرد و مسیری از ریشه تا رسیدن به این گره را به صورت آرایه‌ای از Nodeها درست می‌کند و برای هرکدام actionای که باید انجام دهند تا این مسیر طی شود را مشخص می‌کند و برمی‌گرداند.

متد calculate_state(state, action): یک شیء از جنس State را بر می‌گرداند که حالتی از بازی است که با انجام عملیات action روی حالت state که ورودی‌های این تابع هستند، به آن خواهیم رسید.

نحوه پیاده‌سازی الگوریتم‌ها:

سوال اول:

الگوریتم BFS سوال اول در فایل Q1.py پیاده‌سازی شده‌است، فایل data_structures.py به صورت ds در این فایل import شده‌است تا به توابعش دسترسی داشته باشیم. در تابع main() ابتدا حالت ابتدایی با فراخوانی تابع init_game() با آرگومان **ورودی آدرس فایل تکست حالت اولیه بازی** حالت اولیه را می‌سازیم و سپس یک node از حالت اولیه بدون هیچ والد و با عمق صفر ایجاد می‌کنیم و تابع bfs را با آرگومان ورودی این node فراخوانی می‌کنیم. تابع bfs آرایه solution را که مسیر ریشه تا رسیدن به گره هدف مورد نظر است، به ما برخواهد گرداند که با تابع beauty_print(solution) این مسیر را به همراه تعداد گره‌های تولید شده، تعداد گره‌های بسط داده شده، عمق جواب، عملیات مورد نظر در هر مرحله و حالت بازی در هر مرحله پرینت می‌کند.

```
def main():
    initial_state = ds.init_game("test7.txt")
    initial_node = ds.Node(initial_state, None, None, None, 0)
    solution = bfs(initial_node)
    beauty_print(solution)
```

الگوریتم BFS پیاده‌سازی شده دقیقاً همان الگوریتم موجود در اسلایدهای درس می‌باشد.

```
def bfs(init_node):
    frontier = []
    explored = []
    global nodes_generated_num
    global nodes_expanded_num

    if ds.goal_test(init_node.state):
        return ds.get_solution(init_node)
    frontier.append(init_node)

    while True:
        if frontier == []:
            return 'failure'
        node = frontier.pop(0)
        nodes_expanded_num += 1
        explored.append(node.state)
        actions = ds.get_all_actions(node)
        for act in actions:
            child = ds.Node(node.state, node, None, act, node.cost+1)
            if child.state not in explored and child.state not in [n.state for n in frontier]:
                nodes_generated_num += 1
                if ds.goal_test(child.state):
                    return ds.get_solution(child)
                frontier.append(child)
```

یک لیست frontier و یک لیست explored داریم که در ابتدا خالی هستند. حالت گره اولیه را بررسی می‌کنیم که هدف است یا نه، اگر هدف بود جواب را که همین گره است برمی‌گردانیم. اگر هدف نبود آنرا به لیست frontier اضافه می‌کنیم و در یک حلقه، تا زمانی که لیست frontier خالی نشده و یا به هدف نرسیده ایم، گره اول این لیست را برمی‌داریم و فرزندانش را به ترتیب ایجاد می‌کنیم و **goal_test را موقع تولید فرزندان صدا می‌زنیم** و هرجا به گره هدف رسیدیم، مسیر رسیدن ریشه تا آن گره را برمی‌گردانم و اگر فرزند گره هدف نبود، آنرا به انتهای لیست frontier اضافه می‌کنیم.

سوال دوم:

الگوریتم IDS سوال دوم در فایل Q2.py پیاده‌سازی شده‌است، فایل data_structures.py به صورت ds در این فایل import شده‌است تا به توابعش دسترسی داشته باشیم. در تابع main() ابتدا حالت ابتدایی با فراخوانی تابع init_game() با آرگومان **ورودی آدرس فایل تکست حالت اولیه بازی** حالت اولیه را می‌سازیم و سپس یک node از حالت اولیه بدون هیچ والد و با عمق صفر ایجاد می‌کنیم و تابع ids را با آرگومان ورودی این node فراخوانی می‌کنیم. همچنین حداکثر عمق برای جستجوی ids و عمق اولیه شروع ids به صورت متغیرهای Global در ابتدای فایل Q2.py تعریف شده‌اند و مقادیر آنها قابل تغییر است:

```
4   MAXIMUM_DEPTH = 1000000
5   INITIAL_DEPTH = 0
6
```

نحوه پیاده‌سازی این الگوریتم دقیقاً مطابق اسلایدها می‌باشد و به این صورت است که در ابتدا تابع dls(node, limit) را تعریف کرده‌ام که الگوریتم DLS را تا حداکثر عمق limit با شروع جستجوی DFS از node اجرا می‌کند و اگر تا این عمق به جواب برسد آنرا بر می‌گرداند و در غیر این صورت اگر هنوز گره ای باقی مانده بود فراتر از عمق limit رشته 'cutoff' و اگر هیچ گره دیگری باقی نمانده بود، رشته 'failure' را بر می‌گرداند. **آزمون هدف هم هنگام بسط node اجرا می‌شود** و هرجا به گره هدف رسیدیم، مسیر ریشه تا این گره را برمی‌گرداند. شبه کد الگوریتم DLS پیاده‌سازی شده به صورت 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
```

برای پیاده‌سازی الگوریتم ids(node, initial_depth) هم در یک حلقه با شروع از حداکثر عمق تا INITIAL_DEPTH تا عمق MAXIMUM_DEPTH که هر دو متغیرهای global هستند، الگوریتم DLS را با این محدودیت عمق روی node صدا می‌زنیم.

```
36  def ids(node, initial_depth):
37      global MAXIMUM_DEPTH
38      for limit in range(initial_depth, MAXIMUM_DEPTH):
39          result = dls(node, limit)
40          if result != 'cutoff' and result != 'failure':
41              return result
42
```

سوال سوم:

هیوریستیک ارائه شده برای این سوال:

به ازای هر رنگ، ردیفی را که بیشترین تعداد کارت با آن رنگ وجود دارد شناسایی می‌کنیم و تعداد کارت‌های با آن رنگ را در آن ردیف بدست می‌آوریم. اگر فرض کنیم که در کل از هر کارت n رنگ داشته باشیم و در ردیفی که بیشترین تعداد کارت با یک رنگ مشخص وجود دارد، m تا کارت از این رنگ باشد، حداقل باید n-m کارت باقی‌مانده از آن رنگ که در ردیف‌های دیگر می‌باشند، به این ردیف وارد شوند، یعنی برای اینکه تمام کارت‌های این رنگ در یک ردیف قرار بگیرند، حداقل n-m کارت باید حداقل یکبار جابجا شوند. پس این n-m حد پایین جابجایی برای این رنگ است. به همین ترتیب مجموع این عدد بدست آمده برای تمامی رنگ‌ها را برابر با هیوریستیک گره مورد نظر می‌گیریم.

به عنوان نمونه اگر ۲ رنگ آبی و قرمز داشته باشیم و از هر رنگ ۵ کارت موجود باشد، با فرض اینکه بیشترین تعداد کارت آبی که در یک ردیف قرار دارند، ۳ کارت آبی در ردیف ۱ باشند و بیشترین تعداد کارت قرمز که در ردیفی قرار دارند، ۴ کارت قرمز در ردیف ۲ باشند، حداقل باید 2 = 3 – 5 حرکت برای کارت‌های آبی و 1 = 4 – 5 حرکت برای کارت‌های قرمز انجام دهیم تا همگی آبی‌ها در ردیف ۱ و همگی قرمزها در ردیف ۲ قرار بگیرند.

از آنجایی که این حداقل تعداد حرکاتی است که همه‌ی کارت‌های هر رنگ در یک ردیف قرار بگیرند و نزولی بودن شماره کارت‌ها یا ممکن بودن حرکت کارتی از یک رنگ به طور مستقیم به ردیف مورد نظر که بیشترین تعداد کارت‌های آن رنگ وجود دارند در نظر گرفته نشده، قطعاً مقدار این هیوریستیک کمتر از هزینه واقعی برای رسیدن به حالت هدف خواهد بود و هیوریستیک قابل قبول است.

نحوه پیاده‌سازی:

الگوریتم A* سوال سوم در فایل Q3.py پیاده‌سازی شده‌است، فایل data_structures.py به صورت ds در این فایل import شده‌است تا به توابعش دسترسی داشته باشیم. در تابع main() ابتدا حالت ابتدایی با فراخوانی تابع init_game() با آرگومان **ورودی آدرس فایل تکست حالت اولیه بازی** حالت اولیه را می‌سازیم و سپس یک node از حالت اولیه بدون هیچ والد و با عمق صفر ایجاد می‌کنیم و تابع a_star را با آرگومان ورودی این node فراخوانی می‌کنیم. تابع a_star آرایه solution را که مسیر ریشه تا رسیدن به گره هدف مورد نظر است، به ما برخواهد گرداند که با تابع beauty_print(solution) این مسیر را به همراه تعداد گره‌های تولید شده، تعداد گره‌های بسط داده شده، عمق جواب، عملیات مورد نظر در هر مرحله و حالت بازی در هر مرحله پرینت می‌کند.

نحوه پیاده‌سازی تابع a_star(node) به این صورت بوده است که یک لیست frontier و یک لیست explored داریم که در ابتدا خالی هستند. حالت گره اولیه را بررسی می‌کنیم که هدف است یا نه، اگر هدف بود جواب را که همین گره است برمی‌گردانیم. اگر هدف نبود آنرا به لیست frontier اضافه می‌کنیم و در یک حلقه، تا زمانی که لیست frontier خالی نشده و یا به هدف نرسیده‌ایم، این لیست را بر اساس مقدار f که برابر با همان cost + heuristic (cost در اینجا عمق گره است) به صورت صعودی مرتب می‌کنیم. سپس گره اول این لیست را که کمترین مقدار f را دارد، برمی‌داریم و آزمون هدف را روی آن صدا می‌زنیم تا ببینیم هدف است یا خیر (پس **آزمون هدف هنگام بسط node اجرا می‌شود**) و اگر به هدف رسیدیم، مسیر ریشه تا آن گره را بر

می‌گردانیم. اگر گره مورد نظر هدف نبود، فرزندانش را تولید می‌کنیم و اگر در لیست explored نبودند، آنها را به لیست frontier اضافه می‌کنیم (این لیست در حلقه بعدی براساس مقدار f هر گره مرتب می‌شود). گره اصلی که فرزندانش را تولید کردیم را به لیست explored اضافه می‌کنیم و iteration بعدی انجام می‌شود تا به هدف برسیم یا هیچ گره ای در لیست frontier باقی نماند.

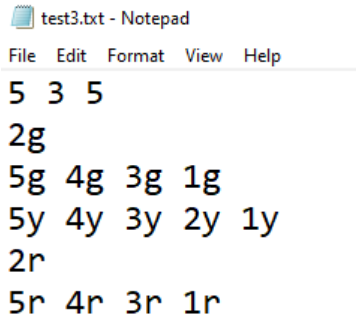
```
8   def a_star(init_node):
9       frontier = []
10      explored = []
11      global nodes_generated_num
12      global nodes_expanded_num
13
14      if ds.goal_test(init_node.state):
15          return ds.get_solution(init_node)
16      frontier.append(init_node)
17
18      while True:
19          frontier.sort(key=lambda x: x.f)
20          if frontier == []:
21              return 'failure'
22          node = frontier.pop(0)
23          nodes_expanded_num += 1
24          if ds.goal_test(node.state):
25                      return ds.get_solution(node)
26          explored.append(node.state)
27          actions = ds.get_all_actions(node)
28          for act in actions:
29              child = ds.Node_H(node.state, node, None, act, node.cost+1)
30              if child.state not in explored and child.state not in [n.state for n in frontier]:
31                  frontier.append(child)
32                  nodes_generated_num += 1
```

مقایسه الگوریتم‌ها از نظر تعداد گره‌های تولیدی، تعداد گره‌های بسط داده شده و عمق جواب‌ها:

با توجه به تست‌هایی که انجام داده‌ام، عمق جواب‌ها در هر سه یکسان می‌باشد (با فرض شروع عمق اولیه الگوریتم IDS از عمق صفر) که مطابق انتظار است؛ چراکه هر سه الگوریتم با فرض اینکه مشابه مسئله ما هدف بهینه هدفی است که در کمترین عمق قرار دارد، پاسخ با کمترین عمق را برمی‌گردانند. یعنی هر سه الگوریتم پاسخ بهینه را بر می‌گردانند.

تعداد گره‌های تولیدی و بسط داده شده در الگوریتم IDS از همه بیشتر است، سپس در الگوریتم BFS بیشترین تعداد گره تولید شده و بسط داده شده و کمترین تعداد گره را هم الگوریتم A* تولید کرده است و بسط داده است. علت این است که الگوریتم IDS شروع می‌کند و از عمق صفر DLS را اجرا می‌کند و عمق را یکی یکی زیاد می‌کند. وقتی جواب در عمق‌های بزرگ وجود داشته باشد این موضوع باعث تولید و بسط تعداد بسیار زیادی node می‌شود. الگوریتم BFS هم بدون استفاده از هیچ هیوریستیکی به طور غیرآگاهانه همه‌ی گره‌ها را به صورت اول سطح شروع به بسط دادن می‌کند که باعث می‌شود در جایگاه دوم قرار بگیرد. الگوریتم A* به طور هوشمندانه گره‌هایی را که شانس بیشتری برای رسیدن به جواب بهینه دارند با توجه به هیوریستیک مورد نظر بسط می‌دهد که باعث می‌شود کمترین تعداد گره تولیدی و بسط داده شده را در بین این سه داشته باشد.

به عنوان مثال، برای تست کیس زیر نتایج اجرای هر سه الگوریتم را با هم مقایسه می‌کنیم:



نتیجه اجرای الگوریتم BFS به صورت زیر می‌باشد که پاسخ در عمق ۶ پیدا شده، ۸۰۲ گره تولید شده و ۳۸۲ گره بسط داده شده است:

```
PS K:\Bachelor\Principles of Artificial Intelligence\Project 1\final>
s of Artificial Intelligence/Project 1/final/Q1.py"
Solution Depth (N): 6, Nodes Generated: 802, Nodes Expanded: 382
```

نتیجه اجرای الگوریتم IDS به صورت زیر می‌باشد که پاسخ در عمق ۶ پیدا شده، ۳۳۸۳۹ گره تولید شده و ۳۳۸۲۸ گره بسط داده شده است:

```
PS K:\Bachelor\Principles of Artificial Intelligence\Project 1\final>
s of Artificial Intelligence/Project 1/final/Q2.py"
Solution Depth (N): 6, Nodes Generated: 33839, Nodes Expanded: 33828
```

نتیجه اجرای الگوریتم A* به صورت زیر می‌باشد که پاسخ در عمق ۶ پیدا شده، ۲۱۱ گره تولید شده و ۶۴ گره بسط داده شده است:

```
PS K:\Bachelor\Principles of Artificial Intelligence\Project 1\final>
s of Artificial Intelligence/Project 1/final/Q3.py"
Solution Depth (N): 6, Nodes Generated: 211, Nodes Expanded: 64
```

همانطور که می‌بینیم، مورد مطرح شده در مورد این مثال هم صادق بوده است.

پاسخ تولید شده توسط الگوریتم A* را هم به عنوان آخرین بحث برای همین تست کیس به صورت زیر می‌باشد:

```
PS K:\Bachelor\Principles of Artificial Intelligence\Project 1\final>
s of Artificial Intelligence/Project 1/final/Q3.py"
Solution Depth (N): 6, Nodes Generated: 211, Nodes Expanded: 64

Depth: 0, Heuristic: 2, F: 2, Action: 2 to 4
1:  2g
2:  5g 4g 3g 1g
3:  5y 4y 3y 2y 1y
4:  2r
5:  5r 4r 3r 1r

Depth: 1, Heuristic: 3, F: 4, Action: 1 to 2
1:  2g
2:  5g 4g 3g
3:  5y 4y 3y 2y 1y
4:  2r 1g
5:  5r 4r 3r 1r

Depth: 2, Heuristic: 2, F: 4, Action: 4 to 2
1:  #
2:  5g 4g 3g 2g
3:  5y 4y 3y 2y 1y
4:  2r 1g
5:  5r 4r 3r 1r
```

```
Depth: 3, Heuristic: 1, F: 4, Action: 5 to 1
1:  #
2:  5g 4g 3g 2g 1g
3:  5y 4y 3y 2y 1y
4:  2r
5:  5r 4r 3r 1r

Depth: 4, Heuristic: 2, F: 6, Action: 4 to 5
1:  1r
2:  5g 4g 3g 2g 1g
3:  5y 4y 3y 2y 1y
4:  2r
5:  5r 4r 3r

Depth: 5, Heuristic: 1, F: 6, Action: 1 to 5
1:  1r
2:  5g 4g 3g 2g 1g
3:  5y 4y 3y 2y 1y
4:  #
5:  5r 4r 3r 2r

Depth: 6, Heuristic: 0, F: 6, Action: None
1:  #
2:  5g 4g 3g 2g 1g
3:  5y 4y 3y 2y 1y
4:  #
5:  5r 4r 3r 2r 1r
```