به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

هوش مصنوعی

گزارش تمرین کامپیوتری یک

نام و نام خانوادگی:

فاطمه شاه حسینی

شماره دانشجویی:

810199440

هدف پروژه : هدف پروژه: پیاده سازی بهینه برخی الگوریتم های جستجو آگاهانه و غیر آگاهان

صورت پروژه : در هر استیت ، باید استیت های ممکن برای مرحله بعد را پیدا کرده و الگوریتم سرچ مدنظر را پیاده کنیم تا بتوانیم پیک را طوری بین پیتزا فروشی ها و دانشجویان حرکت دهیم که همه دانشجویان پیتزای دلخواهشان را دریافت کرده و ترتیب الویت ها هم رعایت شود.

مدلسازی پروژه : در هر مرحله وضعیت صفحه را در یک استیت ذخیره می کنیم و بعد الگوریتم های سرچ را روی استیت ها پیاده می کنیم.

وضعیت هر صفحه شامل شماره state ، پدر استیت، لیست دانشجویان ناراضی، لیست اولویت های رعایت نشده ، پیتزایی که در این استیت در دست پیک است، زمان سپری شده تا الان، یال های لق و نقش هر نود در لحظه می باشد.

```
class State:
    def __init__(self, node_number, parent_node, unsatisfied_students, unhandeled_priorities, holded_pizza, lagh_
      self.node_number = node_number
      self.parent_node = parent_node
      self.unsatisfied_students = unsatisfied_students
      self.unhandeled_priorities = unhandeled_priorities
      self.holded_pizza = holded_pizza
      self.passed_time = passed_time
      self.lagh_edges = lagh_edges
      self.node_role = node_role
    def __str__(self):
      return f"{self.node_number}({self.unsatisfied_students},{self.unhandeled_priorities},{self.holded_pizza})"
```

Initial state: شروع از نود با شماره ورودی، والد -1 و لیست همه دانشجویان، لیست همه اولویت ها، چون پیتزایی در دست ندارد -1 ، زمان 0 ، لیستی از یال های لق و dictionary از هر شماره نود به نقش آن (دانشجو، پیتزافروشی یا خالی) در نقشه.

goal state: استیتی که در آن لیست دانشجویان پیتزا نگرفته (ناراضی) و لیست اولویت های رعایت نشده، خالی باشد.

```
if(state.unsatisfied_students == [] and state.unhandeled_priorities == {}):
  goal_states.append([state, all_states])
  continue
```

Actions: در هر استیت همسایه ها را (به کمک لیست adjacents) شناسایی کرده با توجه به نقش آنها، استیت های بعدی را می سازیم . اگر یالی که قرار است از آن عبور کنیم از یالهای لق بود چک می کنیم چه مقدار اضافه تر باید صبر کنیم و به passed_time اضافه می کنیم.

```
for neighbor in adjacents[state.node_number]:
    wait_time = 1
    child = start
    unsat_stu = copy.deepcopy(state.unsatisfied_students)
    unhandle_pri = copy.deepcopy(state.unhandeled_priorities)
    holded_pz = copy.deepcopy(state.holded_pizza)
    lagh_edg = copy.deepcopy(state.lagh_edges)
    node_rl = copy.deepcopy(state.node_role)
    curr_edge = frozenset({state.node_number, neighbor})
    l_a = lagh_edg.get(curr_edge)
    if( l_a != None):
      latency, available_after = l_a
      if(available_after == 0):
        lagh_edg[curr_edge] = (latancy, state.passed_time + latancy + 1)
      else:
        if (available_after - state.passed_time) > 0:
          wait_time += (available_after - state.passed_time)
```

در پیتزا فروشی اگر دانشجوی الویت داری بر این پیتزا نداشته باشیم و پیتزایی هم از قبل در دست نداشته باشیم ، پیتزایش را برمی داریم و نقش نود را به خالی تغییر می دهیم.(چون هر پیتزا فروشی فقط یک پیتزا ارایه می دهد). در نود های دانشجو، اگر پیتزایی که در دست داریم مال این دانشجوست او را از لیست ناراضی ها حذف می کنیم و اگر این داشنجو بر کسی اولویت داشت، آن الویت را هم از لیست اولویت های اعمال نشده، حذف می کنیم و فیلد پیتزای در دست مان را هم به -1 بر می گردانیم.

```
    if(node_rl[neighbor] == PIZZA):
      if(state.holded_pizza == -1 and state.unhandeled_priorities.get(goal[neighbor]) == None):
        holded_pz = neighbor
        node_rl[neighbor] = EMPTY

    elif(node_rl[neighbor] == STUDENT):
      if(goal.get(state.holded_pizza) == neighbor):
        if neighbor in unsat_stu:
          unsat_stu.remove(neighbor)
        if(neighbor in state.unhandeled_priorities.values()):
          key = {i for i in state.unhandeled_priorities if state.unhandeled_priorities[i]==neighbor}
          for k in key:
            del unhandle_pri[k]
        holded_pz = -1
        node_rl[neighbor] = EMPTY
```

به این صورت استیت جدید ساخته و اگر قبلا explore نشده بود، به لیست frontier ها اضافه می شود.

```
child = State(neighbor, state, unsat_stu, unhandle_pri, holded_pz, lagh_edg, node_rl, state.passed_ti
if(str(child) not in explored and str(child) not in frontier_str):
  all_states += 1
  f.write(str(child)+"\n")
  frontier.append(child)
  frontier_str.add(str(child))
```

الگوریتم های جستجو:

BFS

الگوریتم سرچ ناآگاهانه

در این الگوریتم درخت استیت ها به صورت طبقه طبقه پیمایش می شود. به این معنی که در هر طبقه ابتدا همه فرزندان طبقه بعد برای همه نود ها آن طبقه ساخته شده و سپس به طبقه بعد می رویم.

این الگوریتم به این علت که ابتدا همه ی اعضای یک طبقه را بررسی کره و سپس به طبقه بعد می رود، پاسخ بهینه را تولید می کند. برای پیاده سازی این الگوریتم از queue استفاده می کنیم.

IDS

الگوریتم سرچ ناآگاهانه

در این الگوریتم در هر مرحله یک عمق به تابع dfs می دهیم. تابع دی اف اس از ریشه تا عمق داده شده دی اف اس می زند و بررسی می کند که به مقصد می رسد یا خیر در صورت نرسیدن به استیت هدف، عمق داده شده یک واحد افزایش می یابد.

این الگوریتم به علت اینکه هر دفعه تنها تا عمق گفته شده را برای جواب بررسی میکند و همچنین زمان چک کردن هدف بودن یک استیت که در بالا ذکر شد، جواب بهینه را تولید می کند

*A

الگوریتم سرچ آگاهانه

در این الگوریتم در هر مرحله استیتی که فاصله طی شده تا آن رسیدن به آن به اضافه فاصله تخمین زده شده بین آن استیت و استیت هدف از همه کمتر است انتخاب می شود و فرزندانش به frontier اضافه می شوند. در صورتیکه یکی از فرزندان ساخته از قبل در مرز باشد، اگر مسیر طی شده فعلی از مسیر طی شده قبلی کمتر باشد، جایگزین می شود. در صورتیکه تابع هیوریستیک consistent باشد، پاسخ این الگوریتم بهینه می باشد. برای پیاده سازی این الگوریتم از min heap استفاده می کنیم و لازم است چند فیلد برای محاسبه مقدار تخمین و هزینه مسیری که تا به حال امده ایم به استیت اضافه کنیم و با تابع estimated_cost تخمین را محاسبه کنیم.

```
class State:
    def __init__(self, node_number, parent_node, unsatisfied_students, unhandeled_priorities, holded_pizza, lagh_
      self.node_number = node_number
      self.parent_node = parent_node
      self.unsatisfied_students = unsatisfied_students
      self.unhandeled_priorities = unhandeled_priorities
      self.holded_pizza = holded_pizza
      self.passed_time = passed_time
      self.lagh_edges = lagh_edges
      self.node_role = node_role
      self.estimated_cost = estimated_cost
      self.g = g
      self.f = f
    def __str__(self):
      return f"{self.node_number}({self.unsatisfied_students},{self.unhandeled_priorities},{self.holded_pizza})"
```

(variable) unhandeled_priorities: Any

```
def estimate_cost(state):
    cost = 2 * len(state.unsatisfied_students)
    if goal.get(state.holded_pizza) and goal.get(state.holded_pizza) not in state.unhandeled_priorities:
        cost -= 1
    return cost
```

توضیحات مربوط به تابع هیوریستیک

تابع estimate_cost دو برابر تعداد دانشجویان ناراضی را حساب می کند و اگر دراین استیت پیتزایی در دست پیک بود، یکی از حاصل ضرب قبلی کم می کند و حاصل را به عنوان هزینه تخمینی تا مقصد برمی گرداند. این هیوریستیک consistent است، چون پیک برای رساندن هر پیتزا به مقصد لاقل باید یکبار به نود پیتزافروشی و یکبار به نود دانشجوی مد نظر برود، که با ساده سازی مسیله و فرض اینکه این نود ها به هم وصل اند و نود

اضافه ای برای انتقال لازم نیست ، لاقل دوبرابر تعداد سفارش ها، زمان برای جابه جایی لازم داریم. که اگر نود اولیه پیتزا فروشی باشد یکی از هزینه های رفتن به پیتزا فروشی کم می شود.

*weighted A

الگوریتم سرچ آگاهانه

پیاده سازی این الگوریتم مشابه الگوریتم A* می باشد با تفاوت که ضریبی در تابع هیوریستیک ضرب می شود. این ضریب ممکن است باعث شود که در تابع هیوریستیک دیگر consistant نباشد و در نتیجه همیشه جوابی که از این الگوریتم می گیریم بهینه نمی باشد. به عبارتی در ازای از دست دادن optimality جواب، سرعت رسیدن به جواب را بیشتر می کند. این الگوریتم به این علت که تابع هیوریستیک آن consistent نیست، لزوما پاسخ بهینه را تولید نمی کند. برای پیاده سازی این الگوریتم کافی است ضریب مورد نظر را در خروجی تابع محاسبه کننده هیوریستیک الگوریتم *A ضرب کنیم.

تفاوت ها و مزیت های الگوریتم ها پیاده سازی شده نسبت به هم

*A و الگوریتم های سرچ نا آگاهانه

الگوریتم A* به عنوان یک الگوریتم سرچ آگاهانه، با استفاده از یک تابع هیوریتستیک می تواند پیش بینی کند که چه استیت هایی به هدف نزدیکتر هستند و با اینکار جلوی بسیاری از بررسی های اضافه را بگیرد و از این نظر نسبت به الگوریتم های جست و جوی ناآگاهانه مزیت بالایی دارد. از معایب این روش می تواند گفت که سرعت و درستی محاسبات این الگوریتم به شدت به تابع هیوریستیک آن وابسته می باشد که پیدا کردن آن می تواند کار خیلی سختی باشد.

DFS وBFS

هر دو این الگوریتم ها در صورتی که جواب وجود داشته باشد جواب بهینه را پیدا می کنند. آی دی اس به این علت که استیت های تکراری زیادی را بررسی می کند، سرعت کمتری از بی اف اس دارد. اما مزیت آن نسبت به بی اف اس این است که فضای بسیار کمتری را اشغال می کند. در حالیکه فضای اشغال شده توسط بی اف اس متناسب با تعداد نود های کل درختی است که الگوریتم روی آن اجرا می شود، فضای اشغال شده توسط آی دی اس متناسب با تعداد همسایه های تولید شده در طول راه در حال طی شدن در درخت می باشد.

* A و weighted A*

الگوریتم A* وزن دار با دادن ضریب اهمیت بیشتر به تابع هیوریستیک در محاسبه f، برخی استیت ها که در حالت عادی توسط A* بررسی می شدد را از را به علت فاصله تخمینی تا مقضد بیشتر از رده خارج می کندو در نتیجه نسبت به A* سرعت بیشتری دارد. عیب آن این است که با ضریب دادن به تابع هیوریستیک ، consistent بودن آن از بین می رودو درنتیجه الگوریتم A* وزن دار همیشه پاسخ بهینه را نمی دهد. بلکه پاسخی نزدیک به پاسخ بهینه را می دهد.

توابع کمکی:

با کمک تابع printPath مسیری که پیک باید طی کند تا به استیت نهایی برسد را چاپ می کنیم.

به کمک تابع read_file محتوای ورودی از فایل تکست را می خوانیم و در dictionary ها و لیست های مربوطه ذخیره می کنیم.

جدول تست ها:

Test 1	حداقل زمان لازم برای رساندن پیتزا ها	تعداد کل استیت های دیده شده	میانگین زمان اجرا
BFS	10	189	**0.06**
IDS	10	607	**0.55**
A*	10	114	**0.08**
A* weighted 1	10	101	**0.08**
A* weighted 2	10	42	**0.09**

Test 2	حداقل زمان لازم برای رساندن پیتزا ها	تعداد کل استیت های دیده شده	میانگین زمان اجرا
BFS	21	443	**0.027**
IDS	21	2841	**8.5**
A*	21	398	**0.33**
A* weighted 1	21	386	**0.34**
A* weighted 2	21	306	**0.25**

Test 3	حداقل زمان لازم برای رساندن پیتزا ها	تعداد کل استیت های دیده شده	میانگین زمان اجرا
BFS	31	8399	**5.6**
IDS	31	66826	**440**
A*	31	8091	**7.0**
A* weighted 1	31	7746	**7.5**
A* weighted 2	31	199	**5.3**

همان طور که انتظار داشتیم در تست 2 و3 که پیچیدگی بیشتری دارند ، با وزن دادن به الگوریتم A* ، زمان اجرا کاهش می یابد و تعداد کل استیت های دیده شده در A* وزندار با افزایش ضریب (5 و 1.2) به طرز قابل توجهی کاهش می یابد. انتظار داشتم پاسخ مساله که همان حداقل زمان لازم برای رساندن پیتزا ها بود در A* های وزندار تغییر کند و از حالت بهینه خارج شود که چنین نشد.

نتیجه گیری کلی

در این پروژه الگوریتم های جست و جوی آگاهانه و ناآگاهانه را پیاده سازی کردیم و کارکرد هریک را دیدیم.

هر کدام از این الگوریتم ها ویژگی های خاص خود را دارند که با توجه به محدودیت های مساله ما (مثلا محدودیت در حافظه یا زمان) می توان از آنها استفاده کرد. به طور کلی دیدیم الگوریتم های آگاهانه کارا تر از ناآگاهانه هستند. و اگر سرعت برایمان اهمیت ویژه ای دارد می توان حتی از بهینه بودن جواب هم تا حدودی صرف نظر کرد.

برخی منابع

https://www.educative.io/edpresso/what-is-iterative-deepening-search
https://medium.com/@meghamohan/mutable-and-immutable-side-of-python-c2145cf72747
https://www.researchgate.net/figure/A-search-algorithm-Pseudocode-of-the-A-search-algorithm-operating-with-open-and-closed_fig8_232085273