هوش مصنوعی و سیستم‌های خبره

تمرین سری سوم

مدرس:

دکتر محمدرضا محمدی

طراحان:

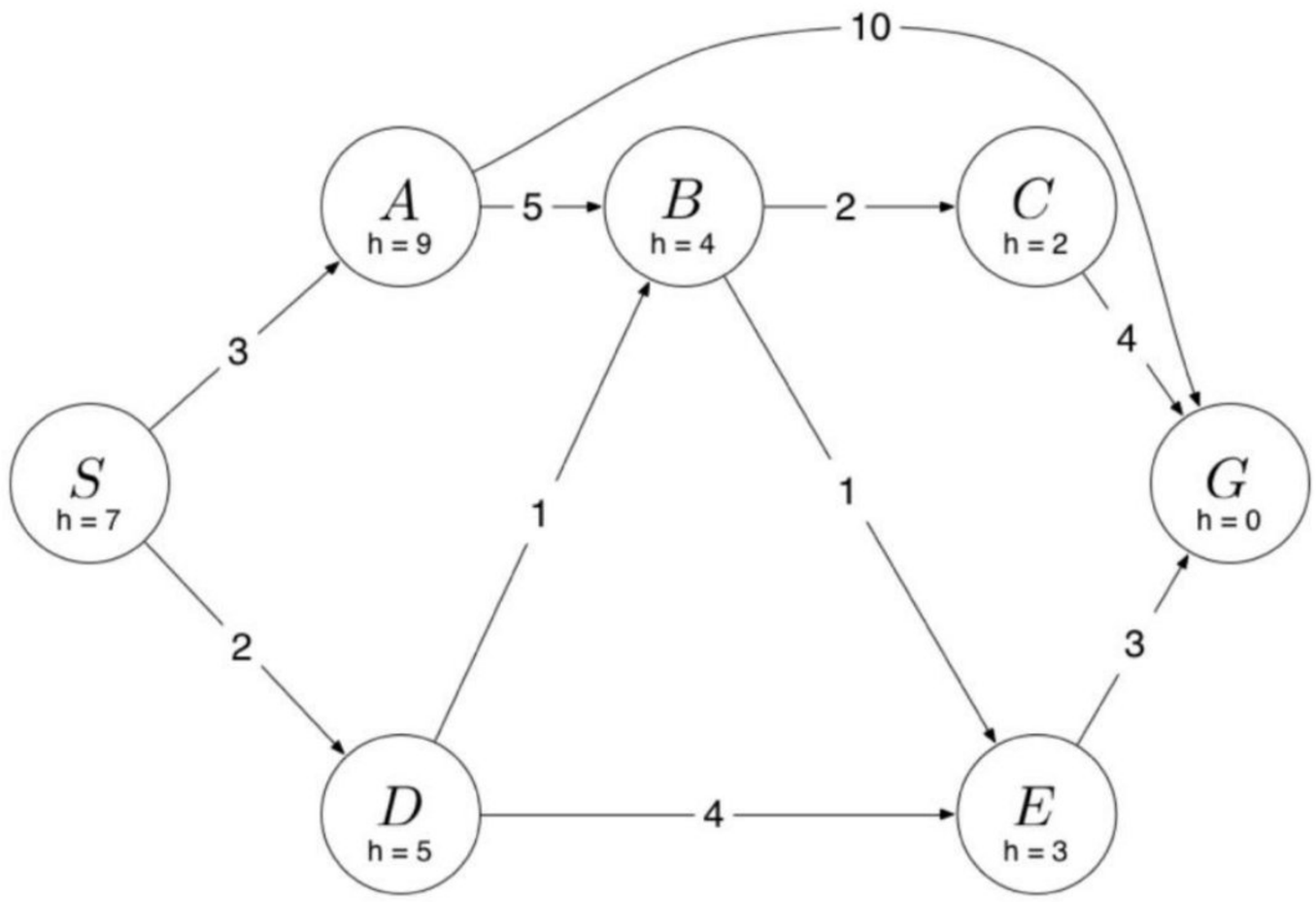
محمد یارمقدم، امیرعلی پاکدامن

مهلت ارسال: ۱۴۰۱/۰۷/۳۰

سوالات تئوری)

سوال یک)

الف) برای گراف نشان داده شده در شکل زیر الگوریتم جستجوی A^* را اجرا نمایید. فرض کنید پیمایش بر اساس ترتیب حروف الفبای انگلیسی انجام میشود. یعنی به طور مثال A -> B -> C قبل از A -> B -> E پیموده میشود.



ب) همین مسئله را با الگورریتم greedy حل نمایید و جواب خود را با حالت قبل مقایسه نموده و مزایا و معایب هر یک را ذکر کنید.

سوال دو)

فرض کنید شما در حال تکمیل بک تابع حریصانه جدید به نام h_1 هستید که در جدول زیر نمایش داده شده است. همه مقادیر جز $h_1(B)$ ثابت هستند.

Node	A	B	C	D	E	F	G
h_1	۱۰	?	۹	۷	۱.۵	۴.۵	۰

برای هر قسمت مجموعه مقادیری که برای $h_1(B)$ مجاز است زا با ذکر توضیح بنویسید.

الف) چه مقادیری از $h_1(B)$ ، h_1 را admissible می‌کنند؟

ب) چه مقادیری از $h_1(B)$ ، h_1 را consistent می‌کنند؟

ج) چه مقادیری از $h_1(B)$ ، باعث می‌شوند الگوریتم جستجوی A^* ترتیب زیر را برای پیمایش طی کند؟

A -> C -> B -> D

سوال پیاده سازی)

سوال سه) هدف در این سوال حل جدول سودوکو به‌وسیله یک عامل هوشمند است.

	1	6	3		8	4	2	
8	4				7	3		
3								
	6		9	4		8		2
	8	1		3		7	9	
9		3		7	6		4	
								3
		5	7				6	8
	7	8	1		3	2	5	

در قسمت اول، هدف حل این جدول با الگوریتم عقبگرد است. الگوریتم عقبگرد بسیار ساده است. این همان رویکردی است که در مسئله n-queen استفاده می شود. شرط اولیه ما این است که یک سلول خالی (که با '0' نشان داده شده است) در جدول پیدا کنیم تا آن را با یک عدد پر کنیم. اگر عامل نقطه خالی پیدا نکرد به این معنی است که جدول پر است و مشکل حل شده است. هر زمان که عامل یک سلول خالی پیدا کند، بررسی می کند که کدام عدد در محدوده 1 تا 9 برای استفاده در سلول بی‌خطر است. پس از یافتن عدد مناسب، سلول را پر می‌کند و دوباره تابع backtracking را فراخوانی می‌کند تا عمیق‌تر در درخت حرکت کند تا سلول بعدی پر شود. این فراخوانی تابع به صورت بازگشتی در هر مرحله انجام می شود تا زمانی که جدول با اعداد پر شود. در هر نقطه اگر نتواند یک سلول را با یک عدد پر کند، به سلول قبلی باز می‌گردد و آن عدد را به انتخاب معتبر دیگری تغییر می‌دهد. کد این بخش:

```
def solveSimpleBackTracking(self):
    location = self.getNextLocation()
    if location[0] == -1:
       return True
    else:
       self.expandedNodes += 1
       for choice in range(1,self.dim+1):
          if self.isSafe(location[0],location[1],choice):
             self.board[location[0]][location[1]] = str(choice)
             if self.solveSimpleBackTracking():
                 return True
             self.board[location[0]][location[1]] = '0'
    return False
```

مابقی کد لازم برای اجرای این کد بر روی یک جدول سودوکو را طبق تابع بالا پیاده‌سازی کنید و اجرا کنید.

در قسمت بعد هدف ایجاد یک الگوریتم *CSP* است. *CSP* مخفف *Constraint Satisfaction Problem* است. بنابراین، هدف اصلی ما برای طراحی چنین الگوریتمی برآورده کردن تمام محدودیت‌های تعریف‌شده‌ای است که مسئله معرفی می‌کند. برای ایجاد یک الگوریتم *CSP*، باید سه ویژگی مسئله خود را نشان دهیم. متغیرها، دامنه ها و محدودیت ها. هر متغیر بخشی از مسئله است که برای حل مشکل باید به مقدار مناسبی نسبت داده شود. دامنه نشان می‌دهد که کدام مقادیر را می‌توان به یک متغیر خاص اختصاص داد. و در نهایت، محدودیت‌ها مشخص

می‌کند که کدام یک از مقادیر موجود در دامنه می‌تواند در لحظه مورد استفاده قرار گیرد. بیایید این تکنیک را روی مسئله سودوکو خود امتحان کنیم.

در مرحله اول، ما به آرایه ای از همه دامنه‌های همه متغیرها نیاز داریم. به عبارت دیگر، یک فضا برای حفظ مقادیر باقیمانده برای هر متغیر مورد نیاز است. بنابراین، یک ویژگی به نام rv به کلاس ما اضافه می شود. در اینجا دامنه مقادیر ثابت بر روی صفحه بازی با ['x'] جابجا شد تا سلول هایی با اعداد ثابت مشخص شود. در موارد دیگر، معیارهای سودوکو بررسی شد تا مقادیر مناسب یک سلول را پیدا و آن را به لیست self.rv اضافه شود:

```
def __init__(self,dim,fileDir):
    self.dim = dim
    self.expandedNodes = 0
    with open(fileDir) as f:
        content = f.readlines()
        self.board = [list(x.strip()) for x in content]
    self.rv = self.getRemainingValues()
def getDomain(self,row,col):
    RVCell = [str(i) for i in range(1 ,self.dim + 1)]
    for i in range(self.dim):
        if self.board[row][i] != '0':
            if self.board[row][i] in RVCell:
                RVCell.remove(self.board[row][i])
    for i in range(self.dim):
        if self.board[i][col] != '0':
            if self.board[i][col] in RVCell:
                RVCell.remove(self.board[i][col])
    boxRow = row - row%3
    boxCol = col - col%3
    for i in range(3):
        for j in range(3):
            if self.board[boxRow+i][boxCol+j]!=0:
                if self.board[boxRow+i][boxCol+j] in RVCell:
                    RVCell.remove(self.board[boxRow+i][boxCol+j])
    return RVCell
def getRemainingValues(self):
    RV=[]
    for row in range(self.dim):
        for col in range(self.dim):
            if self.board[row][col] != '0':
                RV.append(['x'])
            else:
                RV.append(self.getDomain(row,col))
    return RV
```

حال برای این بخش نیز کد های لازم برای اجرا این تابع را پیاده سازی نموده و تست خروجی را گرفته و نتایج را گزارش کنید.

در بخش آخر نتایج دو بخش بالا را با یکدیگر مقایسه نموده و گزارش دهید و راهکارهای خود برای بهبود روش حل را نیز ارائه دهید.

راهنمایی: هدف اصلی در این سوال درک توابع معرفی شده و نوشتن تابع *main* متناسب با آنان است. در قسمت آخر نیز باید یک ایده برای بهبود زمان و فضای موردنیاز برای حل این مسئله معرفی کنید.