

گزارش اول:

الف) کد های 'node.py' شامل تابع های محاسبه مسیر، تابع حرکت و

یک کلاس به نام Node است

1. calculatorHeuristic:

این تابع مقدار هیورستیک بر اساس فاصله منتهی محاسبه میکند.

2. Movement(right,...):

این تابع موقعیت فعلی کبری را بر اساس موقعیت مکانی فعلی و

3: class Node:

این کلاس وضعیت حالت های مختلف را در خود نگه میدارد (Factor Depth Cost)

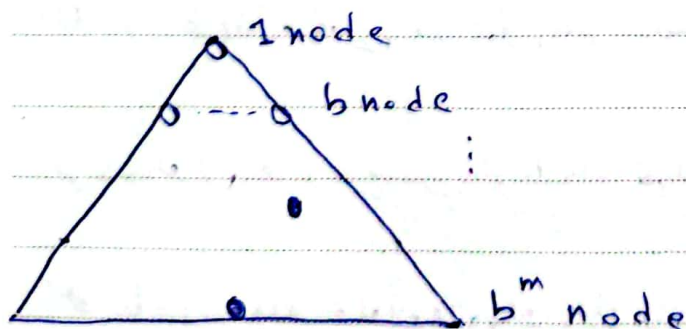
ب) Interface در این پروژه برای نمایش گرافیکی و ارتباط با کاربر استفاده شده است.

1. نمایش گرافیکی:

تصادیر (کبری)، تکیست وکل، ... بلرزاری کند و زمان محاسبه الگوریتم را نشان دهد.

2. تعامل کاربر:

از کاربر با استفاده از pygame ورودی میخواند



۲- الف)

time: $O(b^m)$

space: $O(mb)$

و استفاده از آن در این پروژه نتیجه بهینه به ارفغان من آورد و منطق عمل نمیکند

```
bool IDS (src, target), Max-depth) (ب)
for limit from 0 to max-depth
    if DLS(src, target, limit) == true
        return true
return false

bool DLS (src, target, limit)
    if (src == target)
        return true
    if (limit <= 0)
        return false
    foreach adjacent i of src
        if DLS(i, target, limit)
            return true
    return false
```

DFS را برای ابعاد مختلف با شروع از یک مقدار اولیه فراخوانی می‌کند. در هر فراخوانی،

DFS از خوارشمار رفتن از عمق معین محدود می‌شود در واقع DFS به روش BFS عمل می‌کند

اگر در لول‌های پایین‌تر باشد عملکرد بدتری از DFS دارد زیرا مقدار زیادی دوباره

کار می‌کند و لول‌های او چندین بار بررسی می‌شود.

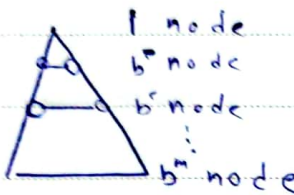
گزارش سوم.

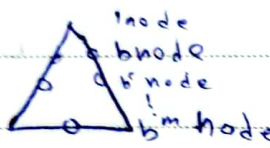
الف) BBFS هر از مبدأ هم از معقد شروع به گسترش دادن

گزینه ها می کنند و در زمان رسیدن آنها به یکدیگر برنامه متوقف

می شود و گزینان ترین مسیر را پیدا می کنند و پس BFS فقط از ابتدا شروع

به گسترش می کنند با این BBFS در مساله ما مفید است.

BFS:  time: $O(b^m)$
space: $O(b^m)$

DFS:  time: $O(b^m)$
space: $O(m \cdot b)$

الگوریتم BFS زمانی عملگر د بهتری دارد هدف در لایه های بالایی باشد چون تمام سطرها را در نظر می گیرد و اگر در لایه های پایین تر باشد کل گره ها را گسترش میدهد و DFS زمانی عملگر د بهتری دارد که هدف در لایه های پایینی باشد زیرا الگوریتم در هر مرحله تا آخرین بهیق گسترش پیدا می کند

گزارش چهارم

الف) الگوریتم Greedy بر اساس هیورستیک ، گره بعدی را گسترش میدهد

با این عملگر د این الگوریتم تنها به هیورستیک بستگی دارد

ب) الگوریتم Greedy همواره به جواب بهینه منتهی شد. در حالت های

استفاده از این الگوریتم بهینه است که $h_m = h^*(n)$ که در آن $k^*(n)$ هزینه

واقعی نزدیک ترین هدف است

الف) به امکان دارد. اگر هزینه یال ها برابر باشد از الگوریتم

UCS به الگوریتم BFS میرسیم و اگر هزینه یال ها به گونه ای باشد

که مسیر مورد نظر یال های آن هزینه صفر و بقیه یال ها هزینه بزرگتر

از صفر داشته باشد از الگوریتم UCS به الگوریتم DFS میرسیم.

ب) مزایا:

بهینه سازی هزینه و پیدا کردن کمترین هزینه و این الگوریتم کامل است.

مسیر با

معایب: از نظر زمانی و پیمایش گره ها و فضای ذخیره سازی نسبت به سایر

الگوریتم ها از معایب آن است.

ج) Dijkstra آینده را در نظر نمیگیرد و بر اساس هزینه یال ها مسیر را

انتخاب میکند برای مثال برای رفتن به مقصد اگر مقصد جلو تر از ما باشد

این الگوریتم فرضی برای یک گام و به جلو و یک گام و به عقب قائل نمیشود
(هزینه برابر)

اما A^* هیورستیک را در نظر گرفته و بر اساس جمع هزینه و هیورستیک

عمل میکند.

(د) به این دلیل به A^* جستجوی گلاخانه میگویند که بادر نظر گرفتن

هیورستیک و هزینه بین یال ها انتخاب میکنند برای مثال اگر مقصد در شمال

باشد بادر نظر گرفتن هیورستیک گره سمت جنوب مبدأ را گسترش میدهند

(ه) قابل قبول: باید هیورستیک گره کوچک تر مساوی هزینه آن گره تا

$$0 \leq h(n) \leq h^*(n) \quad \text{مقصد باشد.}$$

سازگار بودن: در رفتن از یک گره به گره دیگر، تفاضل آن ها باید

کوچک تر یا مساوی هزینه یال بین دو گره باشد

$$h(A) - h(C) \leq \text{cost}(A \text{ to } C)$$

از آنجایی که هیورستیک استفاده شده در این کد حاصله منتهی تا هدف

بوده و کوتاه ترین مسیر است پس برای حالت ها رابطه $h(n) \leq h^*(n)$ برقرار است.

پس از دریافت ستاره Cost بین خونه ها ۵٪ است و ~~تفاوت~~ و تفاضل

هیورستیک آن ها با گره بعدی برابر ۱ است $(\leq 1.5 \times 1)$ بنابراین هیورستیک

۱ استفاده شده در این کد سازگار نیست