CA1 Report

محمد صدرا عباسی **810101469**

۱- نگه داشتن صرفاً map بازی که صرفاً شامل موقعیت فعلی پکمن، روح‌ها، میوه‌ها و دیوارها باشد کافی نیست ایراد اصلی عدم قطعیت است. با چنین تعریفی، هیچ‌گونه پیش‌بینی‌ای از حرکت بعدی روح‌ها نخواهیم داشت برای مثال، یک روح ممکن است در خانه‌ی `(x, y)` باشد و در حال حرکت به سمت مرکز خود باشد، یا در همان خانه‌ی `(x, y)` باشد و در حال حرکت به سمت مخالف باشد. این دو وضعیت، اگرچه از نظر نقشه ظاهراً یکسان هستند، اما منجر به حرکات بعدی کاملاً متفاوتی برای روح می‌شوند. در نتیجه، الگوریتم نمی‌تواند مشخص کند که حرکت (Action) بعدی پکمن safe خواهد بود یا خیر. در نتیجه State باید شامل
وضعیت کامل روح‌ها مانند: موقعیت فعلی `(gx, gy)`/ جهت حرکت فعلی روح/وضعیت روح در چرخه‌ی حرکتش (مثلاً فاصله آنها از مرکز حرکتشان یا یک شمارنده گام).
و وضعیت میوه‌ها: ما باید بدانیم چه میوه‌هایی خورده شده و چه میوه‌هایی باقی مانده‌اند. برای تشخیص Goal State و رعایت شرط مسئله (هیچ میوه‌ی نوع 'B' خورده نمی‌شود تا زمانی که حداقل یک میوه‌ی نوع 'A' باقی مانده باشد)

۲- action چهار حرکت ممکن برای بازیکن است که یک transition function با حرکت ها و state فعلی . تمام قید های ممکن (مانند بررسی عدم تصادم پکمن و روح) را بررسی می کند و state جدید را به طور ضمنی تولید و شبیه سازی می کند که شامل آپدیت کردن موقعیت پکمن , موقعیت روح ها و وضعیت میوه هاست

۳- Initial State همان State است که مستقیماً از فایل نقشه ورودی ساخته می‌شود شامل موقعیت روح و بازی کن و میوه ها و همچنین اطلاعات مربوط به شعاع و مرکز و نوع روح ها . همچنین وضعیت میوه ها به شکل یک آرایه از مقادیر true خواهد بود به این معنی که همه میوه ها در ابتدا وجود دارند.
Goal State هر State ای است که در آن، بخش «وضعیت میوه‌ها»ی آن State، یک آرایه شامل تمام مقادیر `False` باشد یعنی تمام میوه ها خورده شده اند.
Action ها در واقع ۴ حرکت ممکن برای بازی کت هست که یک Transition Function حالت فعلی را می‌گیرند و پس از شبیه‌سازی یک گام زمانی State جدید را برمی‌گردانند.

۴- فضای جستجو را در دو مرحله کاهش می دهیم. ابتدا برای تولید branch ها می توانیم حالت هایی که به fail شدن منجر می شوند شناسایی کرده و آنها را در fringe وارد نکنیم بنابراین از مشتق کردن نودی که در گام بعدی به شکست می رسد جلو گیری کرده ایم. در مرحله بعد , از مشتق شدن نود های تکرار در درخت جستجو جلو گیری می کنیم یعنی با نگه داشتن آرایه ای از تمام نود های قبلی بررسی می کنیم که حالتی می خواهد visit شود حتما حالت جدیدی باشد با عدم چک کردن این موضوع اگر نودی وجود داشته باشد که یک یال ورودی داشته باشد , search دچار infinite loop می شود و به جواب نمی رسد.

۵-

DFS :
از root شروع کرده و در هر گام از الگوریتم یک نود فرزند را expand می‌کند و تا حداکثر عمق ممکن (در آن شاخه) پیش

می‌رود. سپس عقب‌گرد (Backtrack) کرده و نود بعدی را expand می‌کند. دو راه اصلی برای پیاده‌سازی DFS وجود دارد: بازگشتی (Recursive) یا تکراری (Iterative) با استفاده از پشته (Stack).

مزیت اصلی DFS، مصرف حافظه‌ی بهینه‌ی آن است. اما این مزیت، فقط در یک حالت خاص رخ می‌دهد. برای پیاده‌سازی DFS در گراف ، باید نحوه‌ی مدیریت گره‌های تکراری (visited) را مشخص کنیم، که دو حالت دارد:

حالت اول: DFS به روش جستجوی گراف

یک لیست سراسری visited نگه می‌داریم

- **مزیت:** از کاوش تکراری جلوگیری می‌کند و از نظر زمانی بهینه است
- **عیب :** این روش مزیت حافظه‌ی DFS را از بین می‌برد. مجموعه‌ی visited باید تمام گره‌های کاوش‌شده را نگه دارد، بنابراین اوردر حافظه‌ی آن نمایی و برابر با BFS است O(b^d).

حالت دوم: DFS به روش جستجوی درختی

visited را فقط برای مسیر فعلی (از ریشه تا گره فعلی) نگه می‌داریم

- **مزیت:** اوردر حافظه‌ی آن خطی است.
- **عیب:** در گراف‌هایی که مسیرهای متفاوتی به یک گره دارند، دچار کاوش تکراری می‌شود.

جمع‌بندی:

- DFS (در هر دو حالت) جواب بهینه (کوتاه‌ترین مسیر) را تضمین نمی‌کند.
- مزیت اصلی آن (حافظه‌ی خطی) فقط در پیاده‌سازی "Tree Search" (حالت دوم) به دست می‌آید.
- DFS می‌تواند یک جواب (هرچند غیر بهینه) را در صورتی که آن جواب در عمق زیاد باشد، بسیار سریع‌تر از BFS پیدا کند.

:BFS

از گره root شروع کرده و درخت را به صورت سطر به سطر visit می‌کند. در هر گام، قبل از رفتن به سطح بعدی (عمق d+1)، تمام گره‌های سطح فعلی (عمق d) را expand می‌کند. پیاده‌سازی استاندارد و طبیعی BFS به صورت تکراری (Iterative) و با استفاده از یک صف (Queue) انجام می‌شود.
اوردر زمانی BFS نمایی است، در حدود O(b^s) که s عمقی است که حالت جواب در آن پیدا شده است.

عیب : BFS برای تضمین کاوش سطح به سطح، باید تمام گره‌های لایه‌ی مرزی (Frontier) و همچنین تمام گره‌های ملاقات شده (visited) را در حافظه نگه دارد. اوردر حافظه آن همیشه به صورت نمایی O(b^d) است

مزیت : چون BFS گره‌ها را به صورت سطری (بر اساس عمق) بررسی می‌کند، اگر استیتی را پیدا کند که جواب باشد (Goal State)، این جواب قطعاً بهینه (Optimal) خواهد بود

(a
همانطور که اشاره شد، مشکل اصلی DFS عدم تضمین یافتن جواب بهینه است. اما اگر صرفاً بخواهیم جوابی برای مسئله پیدا کنیم (فارغ از بهینه بودن)، می‌توانیم از DFS استفاده کنیم. مزیت اصلی DFS در مصرف حافظه است (این مزیت به شدت به نحوه‌ی پیاده‌سازی آن بستگی دارد)

b)

IDS از جستجوی کم حافظه DFS (یعنی DFS بازگشتی که فقط `visited` مسیر فعلی را نگه می‌دارد) استفاده می‌کند. اما آن را در یک حلقه‌ی بیرونی اجرا می‌کند و به آن محدودیت عمق می‌دهد در نتیجه:
مشکل حافظه‌ی BFS را حل می‌کند: از آنجایی که جستجوی داخلی آن، DFS بازگشتیِ کم‌حافظه است، مصرف حافظه‌ی کلی IDS خطی باقی می‌ماند.

مشکل بهینگی DFS را حل می‌کند: از آنجایی که IDS جستجو را سطح به سطح انجام می‌دهد، اولین جوابی که پیدا می‌کند لزوماً در کمترین عمق ممکن قرار دارد. این کار تضمین می‌کند که جواب پیدا شده (مانند BFS) بهینه (کوتاه‌ترین مسیر) است.

مشکل حلقه‌های DFS را حل می‌کند: محدودیت عمق تضمین می‌کند که DFS در حلقه‌های بی‌نهایت گیر نمی‌افتد.

۶- برای بررسی نتایج time limit را 200 در نظر گرفته و نتایج بدست آمده از 10 مپ را لیست می کنیم:

Algorithm / Map	BFS		DFS		IDS	
	Time	Move number	Time	Move number	Time	Move number
Map 1	0.04	12	0.02	62	22.28	12
Map 2	0.0	2	0.0	2	0.0	2
Map 3	0.3	38	0.11	240	TimeOut	_
Map 4	95.24	238	16.56	3786	TimeOut	_
Map 5	9.25	99	1.82	1380	TimeOut	_
Map 6	8.22	47	1.73	688	TimeOut	_
Map 7	4.12	217	1.14	1071	TimeOut	_
Map 8	167.53	101	11.07	1894	TimeOut	_
Map 9	TimeOut	_	7.89	3097	TimeOut	_
Map 10	TimeOut	_	20.29	7863	TimeOut	_

BFS: در تمام مواردی که موفق شده، مسیر Optimal را پیدا کرده است. اما در نقشه‌های بزرگ به دلیل اوردر نمایی O(b^d) شکست می‌خورد (TimeOut).

DFS: در همه‌ی کیس‌ها جوابی سریع اما غیر بهینه پیدا کرده است .

IDS: در مواردی که سقف زمانی آن تمام نشده ، توانسته دقیقاً مانند BFS مسیر بهینه را پیدا کند. این نشان می‌دهد IDS به درستی مزیت بهینگی BFS را با مزیت حافظه‌ی خطی DFS ترکیب می‌کند، هرچند به قیمت زمان اجرای بسیار بیشتر (به دلیل کاوش تکراری) که منجر به TimeOut در اکثر نقشه‌ها شده است.

۷- زیرا این heuristic در یک سری از موارد از هزینه واقعی بیشتر می شود . مثلا پکمن در A / یک میوه در B / یک میوه در C باشد در این حالت هزینه واقعی برای رسیدن پکمن به هدف A->B + B->C است در حالی که تابع heuristic مقدار A->B +

A->C را محاسبه می کند که این مقدار (بافرض اینکه A->C بزرگتر از B->C باشد) از هزینه واقعی بیشتر است در نتیجه admissible نخواهد بود که شرط لازم برای optimal بود A star است

برای حل مشکل می توان heuristic را همان فاصله منهتن در نظر گرفت اما نسبت به یک میوه هدف نه مجموع همه آنها در این حالت چون این مقدار همیشه کمتر مساوی هزینه واقعی برای رسیدن به تمام میوه هاست پس قطعا admissible خواهد بود

-۸

هیوریستیک ۱ : فاصله منهتن تا نزدیک‌ترین میوه‌ی هدف

۱. تعریف هیوریستیک: این تابع، فاصله‌ی منهتن (dx + dy) از موقعیت فعلی پکمن تا **نزدیک‌ترین** میوه‌ی هدف باقی‌مانده را محاسبه می‌کند.

۲. بررسی Admissibility:

- **Admissible است.**
- **دلیل:** یک هیوریستیک Admissible هرگز هزینه‌ی واقعی (h*) را بیش از حد (Overestimate) برآورد نمی‌کند h(n) <= h*(n).
- هزینه‌ی واقعی (h*)، هزینه‌ی خوردن تمام میوه‌های باقی‌مانده است.
- هیوریستیک ما (h1) فقط هزینه‌ی رسیدن به نزدیک‌ترین میوه را تخمین می‌زند.
- هزینه‌ی واقعی برای خوردن همه‌ی میوه‌ها، قطعاً حداقل به اندازه‌ی هزینه‌ی رسیدن به نزدیک‌ترین میوه خواهد بود

۳. بررسی Consistency:

- **Consistent است.**
- **دلیل:** این هیوریستیک (فاصله‌ی منهتن) این شرط را ارضا می‌کند، زیرا با هر حرکت (هزینه‌ی ۱)، فاصله‌ی منهتن تا نزدیک‌ترین میوه، حداکثر ۱ واحد می‌تواند کاهش یابد.

هیوریستیک ۲ : فاصله منهتن تا دورترین میوه‌ی هدف

۱. تعریف هیوریستیک: این تابع، فاصله‌ی منهتن (dx + dy) از موقعیت فعلی پکمن تا **دورترین** میوه‌ی هدف باقی‌مانده را محاسبه می‌کند.

۲. بررسی Admissibility:

- **Admissible است.**
- **دلیل:** منطق این بخش نیز مشابه h1 است. هزینه‌ی واقعی (h*) هزینه‌ی خوردن تمام میوه‌ها است. پکمن برای تمام کردن کارش، بالاخره باید آن میوه‌ی دورترین را نیز بخورد.
- هزینه‌ی واقعی (که شامل دیوارها، روح‌ها و مسیر بین میوه‌ها می‌شود) قطعاً حداقل به اندازه‌ی فاصله‌ی منهتن (بدون مانع) تا آن دورترین میوه است.

۳. بررسی Consistency:

- **Consistent است.**
- **دلیل:** مانند h1، فاصله‌ی منهتن یک هیوریستیک سازگار است. با هر حرکت (هزینه‌ی ۱)، فاصله‌ی منهتن تا دورترین میوه نیز حداکثر ۱ واحد می‌تواند کاهش یابد و شرط نامساوی مثلث هرگز نقض نمی‌شود.

۹- Time limit = 200

heuristic Map	h1 Time	 move number	h2 Time	 move number
Map 1	0.03	12	0.05	12
Map 2	0.0	2	0.0	2
Map 3	0.25	38	0.23	38
Map 4	90.79	238	83.59	238
Map 5	5.54	99	4.89	99
Map 6	10.79	47	8.43	47
Map 7	6.62	217	6.34	217
Map 8	142.82	101	57.41	101
Map 9	TimeOut	_	TimeOut	_
Map 10	TimeOut	_	175.14	93

با توجه به نتایج هردو هیوریستیک توانسته اند جواب بهینه برای هر map پیدا کنند اما به طور کلی h2 حالت های کمتری را نسبت h1 بررسی می کند و در زمان اجرا سریع تر است این نشان می دهد که h2 هیوریستیک قوی تری نسبت به h1 است.

۱۰- بستگی به پیاده سازی الگوریتم A* دارد

اگر هیوریستیک فقط Admissible باشد، باید پیاده‌سازی پیچیده‌تری داشت که بتواند گره‌ها را مجدداً باز کند در صورتی که مسیر بهتری به آن‌ها پیدا شد، و برای این کار نیاز به نگهداری هزینه در `visited` (مانند Hash Table) دارد. اما اگر هیوریستیک Consistent باشد (که شرط کافی برای Admissible بودنه)، اولین باری که به هر گره می‌رسیم، بهترین مسیر است؛ بنابراین دیگر نیازی به مکانیسم پیچیده‌ی باز کردن مجدد یا نگهداری هزینه در `visited` نیست و یک `set` ساده کافی خواهد بود.
(هیوریستیک‌های ما Consistent هستند).
به طور خاص برای پیاده سازی انجام شده در این پروژه چون مکانیزمی برای Reopening وجود دارد شرط Admissible بودن برای هیوریستیک کافی است و جواب بهینه را پیدا می کند

۱۱- هیوریستیک h2 (ماکزیمم فاصله تا میوه‌ی هدف) قوی‌تر از h1 (مینیمم فاصله) است.

هرچند هر دو هیوریستیک جواب بهینه را پیدا می‌کنند، h2 تخمین نزدیک‌تری به هزینه‌ی واقعی ارائه می‌دهد (h2(n) >= h1(n)). این باعث می‌شود A* فضای حالت کمتری را بررسی کند و در نتیجه زمان جستجوی آن سریع‌تر باشد، همانطور که در نتایج هم مشاهده شد.

۱۲-

Time limit = 250

	MAP1	MAP2	MAP3	MAP4	MAP5	MAP6	MAP7	MAP8	MAP9	MAP10
BFS	0.04	0	0.28	93.52	8.16	10.26	5.69	143.3	222.72	221.08
DFS	0.02	0	0.1	17.32	1.77	1.77	2.06	6.01	9.48	21.02
IDS	20.39	0	TimeOut	TimeOut	TimeOut	TimeOut	TimeOut	TimeOut	TimeOut	TimeOut
A*	0.02	0	0.28	71.19	3.78	8.29	4.78	42.01	219.08	114.29
Weighted A*	0.03	0	0.27	59.95	2.41	7.34	4.82	31.62	166.52	100.24

Data Frames for each map:

1)

Algorithm	Time	Numof Moves	Result
BFS	0.04	12	Success
DFS	0.02	62	Success
IDS	20.39	12	Success
A*	0.02	12	Success
Weighted A*	0.03	12	Success

2)

Algorithm	Time	Numof Moves	Result
BFS	0	2	Success
DFS	0	2	Success
IDS	0	2	Success
A*	0	2	Success
Weighted A*	0	2	Success

3)

Algorithm	Time	Numof Moves	Result
BFS	0.28	38	Success
DFS	0.1	240	Success
IDS	250	0	Timeout
A*	0.28	38	Success
Weighted A*	0.27	38	Success

4)

Algorithm	Time	Numof Moves	Result
BFS	93.52	238	Success
DFS	17.32	3786	Success
IDS	250	0	Timeout
A*	71.19	238	Success
Weighted A*	59.95	240	Success

5)

Algorithm	Time	Numof Moves	Result
BFS	8.16	99	Success
DFS	1.77	1380	Success
IDS	250	0	Timeout
A*	3.78	99	Success
Weighted A*	2.41	99	Success

6)

Algorithm	Time	Numof Moves	Result
BFS	10.26	47	Success
DFS	1.77	668	Success
IDS	250	0	Timeout
A*	8.29	47	Success
Weighted A*	7.34	47	Success

7)

Algorithm	Time	Numof Moves	Result
BFS	5.69	217	Success
DFS	2.06	1071	Success
IDS	250	0	Timeout
A*	4.78	217	Success
Weighted A*	4.82	217	Success

8)

Algorithm	Time	Numof Moves	Result
BFS	143.3	101	Success
DFS	6.01	1894	Success
IDS	250	0	Timeout
A*	42.01	101	Success
Weighted A*	31.62	101	Success

9)

Algorithm	Time	Numof Moves	Result
BFS	222.72	122	Success
DFS	9.48	3097	Success
IDS	250	0	Timeout
A*	219.08	122	Success
Weighted A*	166.52	122	Success

10)

Algorithm	Time	Numof Moves	Result
BFS	221.08	93	Success
DFS	21.02	7863	Success
IDS	250	0	Timeout
A*	114.29	93	Success
Weighted A*	100.24	93	Success