بسمه تعالی

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

هوش مصنوعی

پروژه اول درس

سرچ

محیا قینی

۸۱۰۱۹۶۶۱۵

پاییز ۹۹

فهرست

هدف پروژه

در این پروژه قصد داریم با سه الگوریتم سرچ که به صورت تئوری در کلاس آموختیم، در قالب عملی آشنا شویم. با مقایسه نتایج هر قسمت، می توان خصوصیات هر الگوریتم را با دیگری مقایسه کرد.

شرح مسئله

آشنایی با مسئله

مسئله در حالت کلی بسیار شبیه بازی snake است. با این تفاوت که غذا های مار از قبل رو صفحه قرار دارد و مختصات آن به ما داده شده است. همچنین در صفحه بازی دیوارها برای مار مانع به حساب نمی‌آیند. بدین معنا که مار می توان از دیوارها عبور کرده و از دیوار پایین خارج و از بالا وارد شود و بالعکس. برای دیوارهای چپ و راست هم همینطور است. طول مار در آغاز همیشه یک است و با خوردن غذا یک واحد زیاد می‌شود. دانه ها امتیازات ۱ و ۲ دارند که نشان دهنده تعداد دفعاتی است که مار باید آن را بخورد تا دانه از صفحه محو شود.

دیتا های پروژه

ورودی های مسئله شامل:

۱. سایز صفحه بازی در قالب دو عدد که اولی نمایانگر عرض و دومی طول می‌باشد.

۲. مختصات اولیه مار در قالب دو عدد که اولی نمایانگر عرض اولیه سر مار و دومی طول اولیه سر مار می‌باشد.

۳. تعداد دانه های موجود در صفحه.

۴. به تعداد مورد سوم خط داده میشود در هر کدام سه عدد موجود است، به ترتیب عرض، طول و امتیاز دانه می‌باشد.

نحوه مدل کردن مسئله

برای راحت تر شدن پیاده سازی، از شی گرایی کمک گرفتم. برای هر الگوریتم جزییات پیاده سازی گفته خواهد شد، امّا ابتدا قسمت های مشترک توضیح داده می‌شوند.

۱. کلاس state : مربوط به حالات مختلف که رخ می‌دهد. در هر state مختصات فعلی تمام بدن مار در قالب لیست و دو مجموعه یکی برای دانه های یک امتیازی و دیگری برای دانه های دو امتیازی ذخیره می‌شود. همچنین چندین تابع برای بررسی حالات تکراری داریم.

۲. کلاس snake : بدن مار در این کلاس به شکل یک لیست (زیرا ترتیب اعضا برایمان مهم است) ذخیره می‌شود. همچنین چندین تابع برای پیمایش راحت تر مار داریم.

۳. کلاس board : مربوط به مدیریت صفحه حرکت مار. در تابع init این کلاس عملاً ورودی که گرفته شده اند بررسی می‌شوند، اطلاعات مورد نیاز استخراج می‌شوند و به صورت مناسب ذخیره می‌شوند. همچنین این کلاس شامل تابعی است که تمام حرکت های ممکن که اینجا در چهار جهت هستند را چک کرده با توجه به آن جای جدید سر مار محاسبه می‌کند.

اگر این مختصات جدید با بدن مار تداخل نداشت، یا روی دم قرار نگرفت در حالت طول ۲ و افزایش طول باعث تداخل سر و بدن مار می‌شد، node جدید با توجه به بدن مار جدید و تغییرات احتمالی مجموعه های دانه ها ساخته و برگردانده می‌شود. همچنین یک تابع solve داریم که با توجه به الگوریتم سرچ عملیات متفاوتی انجام می‌دهد. در ادامه این تابع توضیح داده می‌شود.

- Initial state

حالت آغازی شامل مختصات اولیه سر مار و دو مجموعه یکی مختصات دانه های تک امتیازی و دیگری دانه های دو امتیازی که در ورودی آمده‌اند.

- Goal state

حالت نهایی حالتی است که در آن مجموعه های دانه های تک و دو امتیازی خالی باشند و دیگر دانه ای برای خوردن نباشد.

- Action

در این مسئله حرکات شامل چهار جهت بالا و پایین و چپ و راست هستند. این حرکات در هر مرحله طبق شرایط گفته شده در بالا حالات جدیدی برای مسئله ایجاد می‌کنند. همچنین دیوار ها هم مانع نمی‌باشد.

- Node

به طور کلی شامل پدرشان، حرکتی که باعث شده از پدرشان به آنها برسیم و حالت فعلی است. بسته به الگوریتم ممکن است موارد دیگری برای آن ذخیره می‌شود.

الگوریتم های پیاده سازی شده

توضیح الگوریتم ها

برای کم کردن هزینه های چک کردن Node های تکراری explored وfrontier را باهم ادغام کردیم تا تنها حضور در یک set چک شود که سریع تر هم باشد.

← ابتدا این کار را انجام نشد و از نظر زمانی در تست کیس ها با مشکل ایجاد شد.

الگوریتم های سرچ به طور کلی به شرح زیر کار می‌کنند، تنها در ساختمان داده ها و مواردی دیگر تفاوت دارند که هنگام بررسی هر یک به آنها اشاره می‌کنیم:

۱) Node آغازی را می سازد. حالت خود را از ورودی می‌گیرد و پدر و عملیات مربوط به آن خالی می‌باشند.
۲) Node را به frontier اضافه می‌کند.
۳) Node آغازی را به frontier اضافه می‌کنیم.
۴) Node را به explored هم اضافه می‌کنیم.
۵) تا زمانی که به حالت نهایی برسیم یا frontier مان خالی شود:
   - اگر frontier خالی بود پایان.
   - براساس الگوریتم Node را از frontier خارج می‌کنیم.
   - یک لیست از تمام Node هایی که می توان از Node خارج شده به آنها رسید به دست می آوریم.

- به ازای هر عضو لیست بالا:
  - چک می‌کنیم حالت جدید، حالت نهایی مان است یا خیر

بله:

- پایان

خیر:

- این Node به explored اضافه می‌شود.

BFS

در الگوریتم BFS، frontier در اصل یک FIFO می‌باشد و node ای که زودتر به آن اضافه شده است، زودتر هم جهت پیمایش خارج می‌شود. بنابراین کلاس QueueFrontier را ساختیم که بدین شکل از لیست داده خارج کند. این الگوریتم از سطح اول شروع می‌کند و با بررسی تمام فرزندان در عمق جلو می‌رود.

برای بالاتر بردن سرعت چک وجود داده در explored از ساختمان داده set استفاده کردیم.

IDS

الگوریتم IDS همان الگوریتم DFS است امّا با کنترل عمق در هر مرحله. این جمله در ادامه توضیح داده می‌شود امّا اول کمی DFS را بررسی می‌کنیم.

DFS کاملاً مشابه BFS است با این تفاوت که از LIFO برای نگهداری frontier استفاده می‌کند و آخرین چیزی که در آن ذخیره شده را اول بررسی می‌کند. بدین منظور کلاس StackFrontier را پیاده‌سازی کردیم که به شکل توضیح داده شده عمل می‌کند. این الگوریتم از سطح اول شروع می‌کند و هر فرزند را تا عمق می‌پیماید و بعد سراغ فرزند بعدی می‌رود.

امّا IDS؛ این الگوریتم همان DFS است امّا با این تفاوت که برا ی هر فرزند تا عمق آخر نمی‌رود، بلکه تا عمق مشخصی همه Node ها را بررسی می کند سپس عمق را یک واحد زیاد می‌کند. بدین شکل تعداد کلی قدم ها را کاهش می‌دهد.

Explored باز هم یک set است، امّا در آن object هایی از نوع ExploredNode نگهداری می‌شود که در اصل شامل خود state و عمق آن است. این کار بدین دلیل است که از explored برای چک کردن دیده شده ها و نیاز به پیمایش ها استفاده می‌کنیم بنابراین نیاز داریم اگر در عمق بیشتری node تکراری دیدیم آن را به explored دیگر اضافه نکنیم.

A*

الگوریتم A* بسیار شبیه الگوریتم BFS است تنها تفاوت در همان ترتیب پیمایش نگهداری node هاست. این الگوریتم به اصطلاح الگوریتم آگاهانه است، بدین معنی که اطلاعاتی داریم که جست و جو با آگاهی بیشتر باشد. این اطلاعات مجموع هزینه از ابتدا تا node فعلی و تخمین هزینه از node فعلی تا هدف (به اصطلاح heuristic) است. Node ها در frontier به ترتیب این مجموع ذخیره می‌شوند(از کم به زیاد) و به همین ترتیب هم پیمایش می‌شوند. بدین منظور SortedFrontier به کمک کتابخانه heapq پایتون پیاده سازی شد. این کتابخانه داده ها را براساس کلید مشخص شده مرتب می‌کند.

← ابتدا مرتب سازی به صورت دستی انجام شد و از نظر زمانی در تست کیس ها با مشکل ایجاد شد.

هزینه از ابتدا تا اینجا به سادگی قابل محاسبه است. در آغاز هزینه صفر است و به ازای هر عملیات هزینه یک واحد افزایش می‌یابد.

درباره روش تخمین هزینه تا هدف امّا به این سادگی نیست و در ادامه روش برای محاسبه این تخمین توضیح داده خواهد شد.

نحوه تخمین heuristic و بررسی خصوصیات

در این پروژه از دو heuristic استفاده کردیم که هر یک را توضیح می دهیم و خصوصیات مربوط به heuristic را برای آن بررسی می‌کنیم:

۱) ماکسیمم فاصله افقی سر مار در هر مرحله با دانه ها

حالت نهایی حالتی است که همه دانه ها خورده شده باشند. در این heuristic ما در نظر گرفتیم باید از نظر افقی به دورترین دانه برسد در هر مرحله تا در مسیر دانه های نزدیک تر را هم بخورد و به حالت نهایی برسیم. این تخمین دقیق نیست امّا نسبت به دو الگوریتم قبلی آگاهی نسبی به agent از موقعیتی نسبت به حالت نهایی دارد می‌دهد.

دقت شود هنگام محاسبه فاصله، حالت رد شدن از دیوار هم در نظر گرفته شده است و کمترین فاصله تا هر دانه حساب شده است. و سپس بین آنها بیشترین مقدار انتخاب می‌شود.

- بررسی Admissible بودن

Admissible بودن بدین معناست که هزینه ای که ما برای state تا هدف پیش‌بینی می‌کنیم از هزینه واقعی بیشتر نباشد، اگر h(A)، heuristic مرحله A باشد و h*(A) هزینه واقعی حالت A تا هدف باشد:

$$h^*(A) \geq h(A)$$

در این heuristic ما تعداد زیادی از محدودیت ها مانند در نظر گرفتن فاصله عمودی، خوردن دوبار دانه ای دو امتیازی را در نظر نمیگیریم پس به وضوح هزینه ای که تخمین میزنیم از هزینه واقعی کمتر است پس این heuristic، admissible است.

۲) ماکسیمم فاصله افقی به علاوه عمودی سر مار در هر مرحله با دانه ها به علاوه تعداد دانه های مانده (محاسبه فاصله متفاوت برای دانه های دو امتیازی)

در اینجا اطلاعات بیشتری را جهت تخمین در نظر می‌گیریم. اولاً علاوه بر فاصله افقی، فاصله عمودی را هم در نظر میگیریم. همچنین در نظر میگیریم وقتی به دورترین دانه رسید حداقل یک واحد دیگر باید به ازای هر دانه مانده در صفحه هزینه کند. همچنین فاصله را تا دانه های دو امتیازی بدین شکل محاسبه می‌کنیم که وقتی به دانه دو امتیازی رسید حداقل باید به اندازه طولش حرکت کند تا دوباره بتواند به این خانه بیاید و دانه دوم را بخورد و میانگین میگیریم. هنگام محاسبه فاصله، حالت رد شدن از دیوار هم در نظر گرفته شده است و کمترین فاصله تا هر دانه حساب شده است. و سپس بین آنها بیشترین مقدار انتخاب می‌شود.

- بررسی Admissible بودن

در این heuristic در نظر گرفته شده بعد از رسیدن به دورترین دانه به دانه های باقی مانده با تنها یک حرکت می‌توانیم برسیم امّا می‌دانیم در کمترین حالت این هزینه نیاز است بنابراین هزینه ای که تخمین میزنیم کمتر یا حداکثر مساوی هزینه واقعی است که مطابق با تعریف بالاتر Admissible بودن است.

- بررسی Consistent بودن

consistent بودن را بدین شکل تعریف می‌کنیم:

اگر از حالت A به حالت B برویم ،h(A) و h(B) به ترتیب نمایشگر heuristic دو حالت A و B باشد:

$$cost(A\ to\ B) \geq h(A) - h(B)$$

حال اثبات می‌کنیم این heuristic، consistent است.

فرض کنیم دو استیت A و B را داریم که متوالی هستند یعنی از A به B میرویم. می دانیم هزینه این حرکت یک واحد است(یک واحد حرکت مار) و cost(A to B) = 1 و می‌دانیم heuristic را برای هر استیت شکل زیر تعریف می کنیم:

$$heuristic = \max\big(distance(head.seed)\big) + count(seeds) - 1$$

می دانیم در هر مرحله حداکثر سر مار یک واحد به بالا یا پایین یا چپ یا راست می‌پیماید بنابراین فاصله ها تا دانه ها دقیقاً یک واحد تغییر می‌کند. حالت های زیر را بررسی می‌کنیم:

- طی حرکت دانه ای خورده نشود:
  - دانه ای که با آن فاصله ماکسیمم داریم تغییری نکند

در این حالت دانه همان است اما فاصله مان با آن یک واحد کمتر یا بیشتر شده است، بنابراین:

$$h(B) = h(A) \pm 1 \rightarrow$$
$$h(A) - h(B) = \mp 1 \leq 1 \leq cost(A\ to\ B)$$

  - دانه ای که با آن فاصله ماکسیمم داریم تغییری کند

فاصله با دانه ها حداکثر یک واحد زیاد شده است، اگر دانه بخواهد عوض شود فاصله با دانه جدید حداکثر به همان اندازه فاصله با دانه قبلی است وگرنه در مرحله قبل این دانه انتخاب می‌شد، بنابراین:

$$h(B) = h(A) \rightarrow$$
$$h(A) - h(B) = 0 \leq 1 \leq cost(A\ to\ B)$$

- طی حرکت دانه خورده شود:
  - دانه ای که با آن فاصله ماکسیمم داریم تغییری نکند

در این حالت علاوه بر تغییرات بالا یکی از تعداد دانه ها هم کاسته می‌شود، بنابراین:

$$h(B) = h(A) \pm 1 - 1 = \begin{matrix} h(A) \\ h(A) - 2 \end{matrix} \rightarrow$$

$$h(A) - h(B) = \begin{matrix} 0 \\ -2 \end{matrix} \leq 1 \leq cost(A\ to\ B)$$

- دانه ای که با آن فاصله ماکسیمم داریم تغییری کند

در این حالت علاوه بر تغییرات بالا یکی از تعداد دانه ها هم کاسته می‌شود، بنابراین:

$$h(B) = h(A) - 1 \rightarrow$$
$$h(A) - h(B) = 1 \leq 1 \leq cost(A\ to\ B)$$

Weighted A*

این الگوریتم دقیقاً همان الگوریتم A* است تنها یک ضریب α ای در heuristic ضرب می‌شود. این الگوریتم به صورت کد جدا پیاده سازی نشده تنها الگوریتم A* در ورودی یک ضریب میگیرد که همان α است و اگر یک باشد A* عادی است در غیر اینصورت weighted A*. حال تاثیر این کار بر A* را بررسی می‌کنیم.

ایده اصلی این است که ما هزینه کمتر از هزینه واقعی در نظر گرفتیم پس بیاییم در عددی ضرب کنیم تا به هزینه واقعی نزدیک تر شویم. یا به هزینه واقعی نزدیک تر می‌شویم و الگوریتم سریع‌تر میشود امّا ممکن است از هزینه واقعی بالاتر شود و بهینه بودن از بین برود.

در ادامه بعد از اینکه تست کیس های مختلف بررسی شد بیشتر توضیح خواهیم داد.

مقایسه الگوریتم ها

الگوریتم BFS از سطح اول شروع می‌کند و در هر سطح همه فرزندان را به لیست پیمایش اضافه می‌کند و بعد به سطح بعدی می‌رود و این کار را تا زمانی که به حالت نهایی و هدف برسد، ادامه می‌دهد. در نتیجه به حافظه زیادی نیاز دارد تا تمام این سطوح و گره ها را نگه دارد.

در مقابل IDS در هر دور اجرا تنها در هر مرحله یک گره را میگیرد و تا عمق نهایی آن دور آن را پیمایش می‌کند و تناه همان شاخه از گره آغاز تا عمق را نگه می‌دارد. بنابراین IDS نیاز به حافظه کمتری نیاز دارد.

امّا از نظر زمان، BFS تمام گره های هم عمق را همزمان بررسی می‌کند امّا IDS جداگانه گره ها را بررسی میکند. برای مثال تصور کنید جواب در عمق سوم راست ترین شاخه است و IDS تمام شاخه های قبل آن را چک می‌کند و در نهایت به سراغ آن می‌آید امّا BFS عمق سوم همه شاخه ها را همزمان بررسی می‌کند و معمولاً BFS سریعتر از IDS است. ایده اصلی IDS هم همان بوده که به DFS حالت BFS را بتواند بدهد و با آن ترکیب کند.

هر دو این الگوریتم ها بدون آگاهی هستند بدین معنا که agent بدون هیچ اطلاعاتی تمام گره ها را بررسی می‌کند. امّا الگوریتم A* به agent آگاهی نسبی از محیطی که در حال حاضر در آن قرار دهد، می‌دهد.

A* با این آگاهی به agent کمک می‌کند قدم بعدی را منطقی تر و در جهت نزدیک شدن به هدف بردارد و درست است مرتب کردن صف پیمایش آن هزینه بر است امّا این آگاهی (اگر تخمین خوب به آن داده شده باشد) کمک می‌کند طی بررسی قدم های کمتری به پاسخ برسد.

همانطور که گفته شد weighted A* سعی میکند هزینه تخمینی را به هزینه واقعی نزدیک کند و سریعتر شود امّا این کار ممکن است به قیمت از دست دادن بهینه بودن پاسخ نهایی باشد.(اگر هزینه تخمینی حاصل ضرب α از هزینه واقعی بالاتر شود.)

اجرا

در کل سه تست کیس داشتیم ابتدا هر کدام را دوبار اجرا کرده و میانگین زمان اجرا هر کدام را ثبت کرده:

تست اوّل:

الگوریتم	زمان اجرای بار اوّل(s)	زمان اجرای بار دوم(s)	میانگین(s)
BFS	0.113656	0.108672	0.111164
IDS	0.566486	0.556537	0.5615115
A^*_1	0.055889	0.055914	0.0559015
A^*_2	0.021979	0.022977	0.022478
Weighted $A^*_1(\alpha=1.8)$	0.037911	0.031914	0.0349125
Weighted $A^*_1(\alpha=4)$	0.007977	0.008005	0.007991
Weighted $A^*_2(\alpha=1.8)$	0.032916	0.033909	0.0334125
Weighted $A^*_2(\alpha=4)$	0.002035	0.002990	0.0025125

تست دوم:

الگوریتم	زمان اجرای بار اوّل(s)	زمان اجرای بار دوم(s)	میانگین(s)
BFS	2.712742	2.735626	2.724184
IDS	6.156550	6.055900	6.106225
A^*_1	0.407910	0.338097	0.3730035
A^*_2	0.046899	0.028933	0.037916
Weighted $A^*_1(\alpha=1.8)$	0.043849	0.041926	0.0428875
Weighted $A^*_1(\alpha=4)$	0.073776	0.073805	0.0737905
Weighted $A^*_2(\alpha=1.8)$	0.036862	0.033941	0.0354015
Weighted $A^*_2(\alpha=4)$	0.369034	0.447805	0.4084195

تست سوم:

الگوریتم	زمان اجرای بار اوّل(s)	زمان اجرای بار دوم(s)	میانگین(s)
BFS	41.467034	41.222706	41.34487
IDS	51.293819	49.608524	50.4511715
A^*_1	3.921507	3.838731	3.880119
A^*_2	0.951712	0.930580	0.941146
Weighted $A^*_1(\alpha=1.8)$	1.327466	1.478085	1.4027755
Weighted $A^*_1(\alpha=4)$	0.262358	0.269341	0.2658495
Weighted $A^*_2(\alpha=1.8)$	0.199467	0.169506	0.1844865
Weighted $A^*_2(\alpha=4)$	0.058841	0.060817	0.059829

حال با توجه به زمان های بالا جداول زیر را پر می‌کنیم و در ستون زمان اجرا همان میانگین به دست آمده در بالا را می‌گذاریم:

تست اوّل:

الگوریتم	فاصله جواب	مسیر جواب	تعداد استیت دیده شده	تعداد استیت مجزا دیده شده	زمان اجرا
BFS	12	DLUULULLUULL	9840	4867	0.111164
IDS	12	LDRRDDRRDRDD	48184	11595	0.5615115
A^*_1	12	LDLUUULLLUUR	3553	2155	0.0559015
A^*_2	12	LDDDRRRRDDDR	1301	932	0.022478
Weighted $A^*_1(\alpha=1.8)$	12	LDLURDUULULL	2074	1254	0.0349125
Weighted $A^*_1(\alpha=4)$	14	URURDRDRDLLLLL	511	364	0.007991
Weighted $A^*_2(\alpha=1.8)$	14	URURRDRDDRRRRR	1814	1196	0.0334125
Weighted $A^*_2(\alpha=4)$	14	URURRDDDRRRRRR	135	126	0.0025125

تست دوم:

الگوریتم	فاصله جواب	مسیر جواب	تعداد استیت دیده شده	تعداد استیت مجزا دیده شده	زمان اجرا
BFS	15	UDRLLUUUULULLLL	107914	47993	2.724184
IDS	15	RLLRULLUUULLLLU	525368	94576	6.106225
A^*_1	15	RLLURUULLUULLLL	19867	11639	0.3730035
A^*_2	15	RULLDLUUUUULLLL	1525	1276	0.037916
Weighted $A^*_1(\alpha=1.8)$	15	RLLURUUULLLLULL	2605	1862	0.0428875
Weighted $A^*_1(\alpha=4)$	15	RLLURUUULLULLLL	4536	2646	0.0737905
Weighted $A^*_2(\alpha=1.8)$	19	ULDRRUUUUULLLDLULLL	1884	1430	0.0354015
Weighted $A^*_2(\alpha=4)$	19	ULDRRUUUUULLLDLULLL	21028	9964	0.4084195

تست سوم:

الگوریتم	فاصله جواب	مسیر جواب	تعداد استیت دیده شده	تعداد استیت مجزا دیده شده	زمان اجرا
BFS	25	URDDDRDRRDDRRRURRDLLUULLL	456480	203650	41.34487
IDS	25	RURDDDRRDRRRDDRRULLDLLLUU	3856272	497732	50.45117
A^*_1	25	RUDDDRRDRRRDRDRRULLDLUULL	159187	76818	3.880119
A^*_2	25	RUDDDRDRRRDDRRRRULLDLUULL	45216	23099	0.941146
Weighted $A^*_1(\alpha=1.8)$	25	RUDDRDRDRDDRRRURRDLLULULL	67866	33704	1.402775
Weighted $A^*_1(\alpha=4)$	26	URDDDRDRRDLURDRRDRURRRDLLL	14649	8145	0.265849
Weighted $A^*_2(\alpha=1.8)$	26	RUDDRDRDRDLURRRDDRURRDLDLL	9249	4946	0.184486
Weighted $A^*_2(\alpha=4)$	26	URDDDRRDRRDLURRDRDRURDDLLL	3532	2014	0.059829

مقایسه Heuristic ها

همانطور که در جداول اجرا بالا قابل مشاهده است Heuristic دوم که سعی می‌کند با استفاده از اطلاعات بالاتر تخمین بهتری از هزینه تا هدف بزند، تعداد استیت های مجزا و غیر مجزا کمتری نسبت به Heuristic دیگر می‌بیند. همچنین سریع‌تر هم هست.

بررسی Weighted A*

همانطور که در جداول اجرا بالا قابل مشاهده است این الگوریتم کمک کرده است زمان اجرا بسیار کمتر شود و هرچه α بزرگتر هم باشد این کاهش زمان اجرا قابل مشاهده تر است.

اما میبینیم در مواردی به دلیل آن که با ضرب α در تخمین هزینه، تخمین را از واقعیت بزرگتر کردیم دیگر admissible نیست و جواب بهینه حاصل نمی‌شود.

نتیجه گیری

هر سه الگوریتم BFS، IDS، A*(به شرط داشتن heuristicای که admissible و consistent باشد) بهینه هستند. BFS تمام سطوح را همزمان می‌پیماید و به عمق می‌رود بنابراین بهینه است. IDS هم با محدود کردن عمق در هر مرحله الگوریتم DFS را به یک الگوریتم بهینه تبدیل می‌کند.

اما از نظر زمانی با تخمین مناسب A* میتواند بسیار سریع جستجو را انجام دهد و تعداد استیت های بسیار کمی را ببیند. همچنین IDS بیشترین تعداد استیت را می‌بیند و کندترین الگوریتم است.

اگر برایمان سرعت بسیار اهمیت داشته باشد و بهینه بودن جواب حائز اهمیت بالایی نباشد می‌توانیم از weighted A* هم استفاده کنیم.