

به نام خدا



درس هوش مصنوعی و سیستم های خبره

تمرین سوم

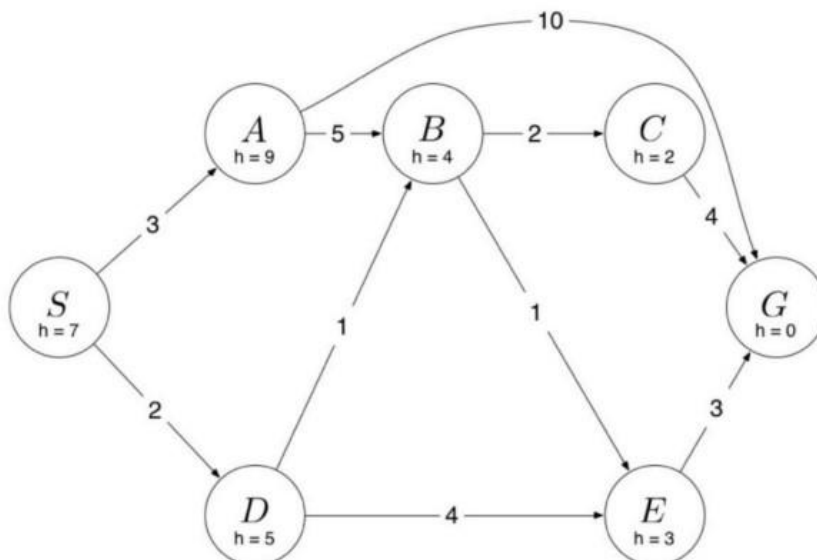
مدرس : دکتر محمدی

دانشجو : سارا سادات یونسی / ۹۸۵۳۳۰۵۳

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 $A \rightarrow B \rightarrow C$ قبل از $A \rightarrow B \rightarrow E$ پیموده میشود.



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

توضیح سوال اول - ۱

الف) همانگونه که می دانیم الگوریتم A^*

نحوه ی EXPAND کردن نود ها در A^* با منطق زیر اجرا میشود :

Expand a node n most likely to be on an optimal path

Expand a node n the cost of the best solution through n is optimal

Expand a node n with lowest value of $g(n) + h^*(n)$

- $g(n)$ is the cost from root to n

- $h^*(n)$ is the optimal cost from n to the closest goal

We seldom know $h^*(n)$ but might have a heuristic approximation $h(n)$

$A^* = \text{tree search with priority queue ordered by } f(n) = g(n) + h(n)$

تابع هیوریستیک ما هم admissible است هم constant است :

در نتیجه با توجه به نکات بالا EXPAND می کنیم :

برای $h1$ خواهیم داشت : در هر مرحله $f(n)=g(n)+h(n)$ را چک می کنیم و نود با $f(n)$ کمتر را انتخاب می کنیم که $g(n)$ value یال ما می باشد و $h(n)$ value heuristic هر نود ما میباشد :

$$\begin{array}{l} 1 \\ S(0+7=7) \rightarrow A(3+9=12) \\ 2 \quad D(2+5=7) \rightarrow E(6+3=9) \\ 3 \quad B(3+4=7) \rightarrow C(5+2=7) \rightarrow G(9+0=9) \\ 4 \\ 5 \quad E(4+3=7) \rightarrow G(7+0=7) \quad 6 \end{array}$$

ابتدا از node s شروع می کنیم دو انتخاب داریم با توجه به اینکه مقدار تابع در d کمتر است d را پیمایش می کنیم و A را رها می کنیم حالا از d دو انتخاب داریم که با توجه به اینکه مقدار تابع در b کمتر است b را پیمایش می کنیم و e را رها حالا از b دو انتخاب داریم c و e که هر دو در نهایت دارای $f(n)=7$ هستند با توجه به فرض مسئله ابتدا c را پیمایش می کنیم و سپس به g می رویم اما با رفتن e به g در نهایت مقدار تابع کمتری خواهیم داشت .

در نتیجه

Path : $S \rightarrow D \rightarrow B \rightarrow E \rightarrow G$

Expand : $S \rightarrow D \rightarrow B \rightarrow C \rightarrow E \rightarrow G$

Highest priority = lowest Cost

This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue.

STEP 1 : $S(7)$

STEP 2: $D(7)$, $A(12)$

STEP3 : $B(7)$, $E(9)$

STEP 4: C (7) , E(7)

STEP 5: G (9) , E(7)

STEP 6: G (7) STOP ALGORITHM

همانگونه که گفته شد ابتدا ما یک آرایه خالی برای نگه داری مقدار تابعی خود که دارای $f(N)$ تابع است در نظر گرفته سپس فرانتیر و PRIORITY خود را با توجه به آن می چینیم که میتواند در هر مرحله اپدیت شود و به آرایه اضافه شود اگر نسبت به قبل $F(N)$ کمتر داشته باشیم و در غیر این صورت REJECT می شود و اگر $F(N)$ مقدار بزرگتری از مقدار تابعی جدید داشته باشد یا مقدار جدید جایگزین می شود.

(ب) همانگونه که میدانیم نحوه ی عملکرد و policy جست و جوی حریصانه به این صورت است :

Strategy: expand a node that you think is closest to a goal state

- Heuristic: estimate of distance to nearest goal for each state

A common case: - Best-first takes you straight to the (wrong) goal

Worst-case: like a badly-guided DFS

بنابراین نود ها با کمترین هیوریستیک را انتخاب می کنیم :

$S \rightarrow D \rightarrow E \rightarrow G$

ابتدا از S شروع می کنیم چون هیوریستیک d کم تر است ان را انتخاب می کنیم سپس بین دو نودی که در فرانتیر ما هستند یعنی e , b چون هیوریستیک e کم تر است ان را انتخاب کرده و سپس به g می رویم.

مقایسه و معایب و مزایا :

همانطور که میبینیم در الگوریتم greedy یک نود کمتر از A^* پیمایش کردیم اما در نهایت جواب A^* بدلیل در نظر گرفتن هزینه مسیر ها بهینه تر خواهد بود بدلیل در نظر گرفتن $g(n)+f(n)$ و در الگوریتم گریدی سریع تر به جواب میرسیم چون تعداد نود های پیمایش شده کمتر است.

مزایای greedy :

۱. پیاده سازی رویکرد حریصانه آسان است.

معمولا پیچیدگی زمانی کمتری دارند.

الگوریتم های حریص را می توان برای اهداف بهینه سازی یا یافتن نزدیک به بهینه سازی در صورت بروز مشکلات سخت مورد استفاده قرار داد.

معایب greedy:

راه حل بهینه محلی ممکن است همیشه در سطح global بهینه نباشد.

مزایای A^* :

کامل و بهینه است.

این بهترین تکنیک در بین تکنیک های دیگر است. برای حل مسائل بسیار پیچیده استفاده می شود.

* A این بهینه کارآمد است، یعنی هیچ الگوریتم بهینه دیگری تضمین شده برای گسترش گره های کمتر از وجود ندارد.

این الگوریتم جستجوی بهینه از نظر اکتشافی است.

این یکی از بهترین تکنیک های جستجوی اکتشافی است.

برای حل مشکلات پیچیده جستجو استفاده می شود.

هیچ الگوریتم بهینه دیگری تضمین شده برای گسترش گره های کمتر از A^* وجود ندارد

معایب A^* :

این الگوریتم در صورتی کامل می شود که **branching factor** محدود باشد و هر عمل دارای هزینه ثابت باشد.

استفاده می شود، $h(n)$ * به شدت به دقت الگوریتم اکتشافی که برای محاسبه A سرعت اجرای جستجوی بستگی دارد.

مشکلات پیچیدگی دارد.

مقایسه ی A^* و الگوریتم greedy:

هر دو الگوریتم در دسته الگوریتم‌های «بهترین جستجوی اول» قرار می‌گیرند، که الگوریتم‌هایی هستند که می‌توانند هم از دانش کسب‌شده در حین کاوش در فضای جستجو، که با $g(n)$ نشان داده می‌شود و هم از یک تابع اکتشافی که با $h(n)$ نشان داده می‌شود، استفاده کنند.

هر یک از این الگوریتم‌های جستجو یک «evaluation function» را برای هر گره n در نمودار (یا فضای جستجو) تعریف می‌کند که با $f(n)$ نشان داده می‌شود. این تابع ارزیابی برای تعیین اینکه کدام گره، در حین جستجو، ابتدا "expand" شده است، استفاده می‌شود، یعنی کدام گره ابتدا از "frontier" حذف می‌شود تا "visit" شود. فرزندان به طور کلی، تفاوت بین الگوریتم‌های دسته «بهترین-اول» در تعریف تابع ارزیابی $f(n)$ است.

در مورد الگوریتم BFS حریص، تابع ارزیابی $f(n)=h(n)$ است، یعنی الگوریتم greedy BFS ابتدا گره ای را که فاصله تخمینی آن تا هدف کوچکترین است، گسترش می‌دهد. بنابراین، greedy BFS از "دانش گذشته"، یعنی $g(n)$ استفاده نمی‌کند. از این رو معنای آن "greedy" است. به طور کلی، الگوریتم حریصانه BST کامل نیست، یعنی همیشه این خطر وجود دارد که مسیری را طی کنید که به هدف نمی‌رسد. در الگوریتم حریصانه BFS، تمام گره‌های حاشیه (یا حاشیه یا مرز) در حافظه نگهداری می‌شوند و گره‌هایی که قبلاً گسترش یافته اند نیازی به ذخیره در حافظه ندارند و بنابراین می‌توان آنها را دور انداخت. به طور کلی، BFS حریص نیز بهینه نیست، یعنی مسیر یافت شده ممکن است مسیر بهینه نباشد. به طور کلی، پیچیدگی زمانی $O(bm)$ است، جایی که b ضریب انشعاب (حداکثر) و m حداکثر عمق درخت جستجو است. پیچیدگی فضا با تعداد گره‌ها در حاشیه و با طول مسیر یافت شده متناسب است.

در مورد الگوریتم A^* ، تابع ارزیابی $f(n)=g(n)+h(n)$ است که h یک تابع اکتشافی قابل قبول است. به این واقعیت اشاره دارد که A^* از یک تابع اکتشافی قابل قبول استفاده می‌کند، که اساساً به این معنی است که A^* بهینه است، یعنی همیشه مسیر بهینه را بین گره شروع و گره پیدا می‌کند. گره هدف A^* نیز کامل است (مگر اینکه تعداد بی‌نهایت گره برای کاوش در فضای جستجو وجود داشته باشد). پیچیدگی زمانی $O(bm)$ است. با این حال، A^* باید در حین جستجو، همه گره‌ها را در حافظه نگه دارد، نه فقط گره‌هایی که در حاشیه هستند، زیرا A^* اساساً یک «جستجوی جامع» را انجام می‌دهد (که «آگاه است»، به این معنا که از یک تابع اکتشافی استفاده می‌کند.).

به طور خلاصه، BFS حریص کامل نیست، بهینه نیست، دارای پیچیدگی زمانی $O(bm)$ و پیچیدگی فضایی است که می تواند چند جمله ای باشد A^* کامل، بهینه است و پیچیدگی زمانی و مکانی آن $O(bm)$ است. بنابراین، به طور کلی، A^* بیشتر از BFS حریص از حافظه استفاده می کند. وقتی فضای جستجو بسیار زیاد باشد، A^* غیر عملی می شود. با این حال، A^* همچنین تضمین می کند که مسیر یافت شده بین گره شروع و گره هدف، مسیر بهینه است و در نهایت الگوریتم خاتمه می یابد. از طرف دیگر، Greedy BFS از حافظه کمتری استفاده می کند، اما تضمین های بهینه و کامل بودن A^* را ارائه نمی دهد. بنابراین، اینکه کدام الگوریتم "بهترین" است بستگی به زمینه دارد، اما هر دو "بهترین" - اولین جستجو هستند

سوال دو)

فرض کنید شما در حال تکمیل یک تابع حریمانه 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 را با ذکر توضیح بنویسید.

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

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

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

$A \rightarrow C \rightarrow B \rightarrow D$

پاسخ سوال دو

الف) همانگونه که می دانیم $admissibility$

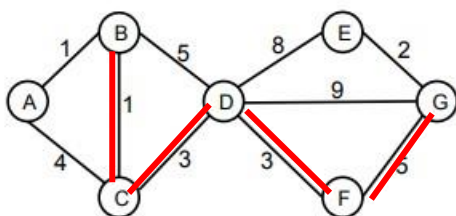
به این معناست که مقدار $huristic$ تخمین زده شده از مقدار $huristic$ واقعی کمتر مساوی باشد ، مناسب و مورد تایید می باشد.

$$h(n-1) \leq h^*(n-1)$$

برای اینکه $h_1(B)$ $admissible$ باشد، $h_1(B)$ باید کمتر یا مساوی هزینه بهینه واقعی از B تا هدف باشد.

با توجه به شکل و مسیر نشان داده شده روی شکل مجموع هزینه از B تا G در بهینه ترین و کم ترین حالت ۱۲ است.

هزینه مسیر B-C-D-F-G



مجموع هزینه یال ها : $12 = 5 + 3 + 3 + 1$

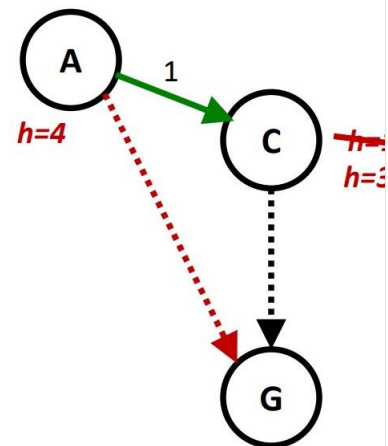
در نتیجه:

$$0 \leq h_1(B) \leq 12$$

(ب) همانگونه که در توضیح سازگاری در اسلایدها داشتیم :

Consistency of Heuristics

- Main idea: estimated heuristic costs \leq actual costs
 - Admissibility: heuristic cost \leq actual cost to goal
 $h(A) \leq h^*(A)$
 - Consistency: heuristic "arc" cost \leq actual cost for each arc
 $h(A) - h(C) \leq c(A, C)$
or $h(A) \leq c(A, C) + h(C)$ (triangle inequality)
 - Note: h^* necessarily satisfies triangle inequality
- Consequences of consistency:
 - The f value along a path never decreases:
 $h(A) \leq c(A, C) + h(C) \Rightarrow g(A) + h(A) \leq g(A) + c(A, C) + h(C)$
 - A^* graph search is optimal



با توجه به تعریف باید $h_1(B)$ به اضافه ی arc یا همان عدد یال ها بزرگتر مساوی عدد هیوریستیک نود های در ارتباط با آن باشند .

همه ی نود ها به جز B شرط $consistent$ بودن را دارا هستند . حال به بررسی حالت هایی که شامل نود B می شوند می پردازیم :

$$h(A) \leq c(A, B) + h(B) \rightarrow h(B) \leq c(B, A) + h(A)$$

$$h(C) \leq c(C, B) + h(B) \rightarrow h(B) \leq c(B, C) + h(C)$$

$$h(D) \leq c(D, B) + h(B) \rightarrow h(B) \leq c(B, D) + h(D)$$

پس از جایگذاری اعداد خواهیم داشت :

$$9 \leq h_1(B) \leq 10$$

ج) همانگونه که می دانیم الگوریتم A^*

نحوه ی EXPAND کردن نود ها در A^* با منطق زیر اجرا میشود :

Expand a node n most likely to be on an optimal path

Expand a node n the cost of the best solution through n is optimal

Expand a node n with lowest value of $g(n) + h^*(n)$

- $g(n)$ is the cost from root to n

- $h^*(n)$ is the optimal cost from n to the closest goal

We seldom know $h^*(n)$ but might have a heuristic approximation $h(n)$

$A^* = \text{tree search with priority queue ordered by } f(n) = g(n) + h(n)$

بنابراین ما traverse کردن را از A شروع می کنیم و دو انتخاب خواهیم داشت یا B یا C با توجه به ساختار جست و جوی A^* که کمترین میزان تابع را بر می گرداند باید کمترین $h(n) + g(n)$ یا بهینه ترین آن ها را انتخاب کند در نتیجه چون باید بعد از A , C انتخاب شود می گوییم

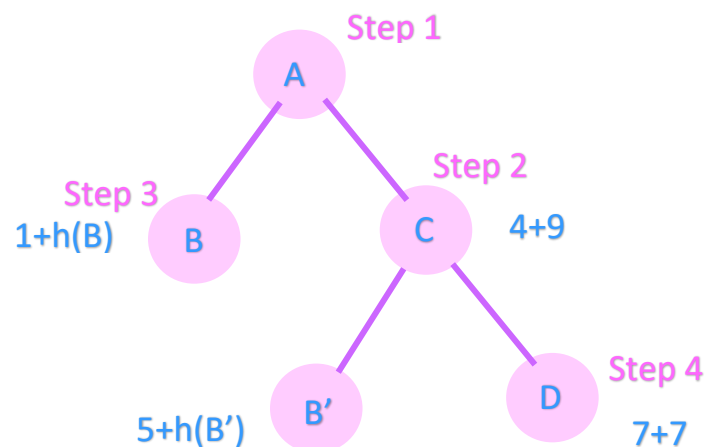
$$f(C) < f(B) \rightarrow 1+h(B) > 4+9 \rightarrow h(B) > 12$$

در ادامه می توان از C به B یا D رفت اما در پیمایش از ما خواسته شده تا به نود B برگردیم پس باید تابع B در اینجا بهینه تر از D باشد پس در این صورت می گوییم

$$f(B') < f(D) \rightarrow 1+h(B') < 4+3+7 \rightarrow h(B') < 13 \rightarrow h(B) > 13$$

در نتیجه با اشتراک گیری از این دو مورد به این نتیجه میرسیم که

$$12 < h(B) < 13$$



گزارش کلی کد ها

ابتدا به بررسی کلی الگوریتم **backtracking** :

Naive Approach

رویکرد ساده لوحانه این است که تمام **config** ممکن از اعداد از ۱ تا ۹ را برای پر کردن سلول های خالی ایجاد کنیم. هر **config** را یکی یکی امتحان کنی تا **config** صحیح پیدا شود، یعنی برای هر موقعیت اختصاص داده نشده، موقعیت را با یک عدد از ۱ تا ۹ پر کنیم. اگر پرینت ایمن برای موارد دیگر تکرار شود

برای حل مشکل مراحل زیر را دنبال می کنیم:

تابعی ایجاد کنید که بررسی کند آیا ماتریس داده شده سودوکو معتبر است یا نه. **hashmap** را برای سطر، ستون و کادرها نگه دارید. اگر هر عددی دارای فرکانس بیشتر از ۱ در **hashmap** باشد، **false** را برگردانید، در غیر اینصورت **true** را برگردانید.

یک تابع بازگشتی ایجاد کنید که یک **grid** و شاخص سطر و ستون فعلی را بگیرد.

برخی از موارد پایه را بررسی کنیم:

اگر شاخص در انتهای ماتریس است، یعنی $i=N-1$ و $j=N$ ، سپس بررسی کنید که آیا **grid** ایمن است یا نه، اگر ایمن است **grid** را چاپ کنید و **true** را برگردانید، در غیر این صورت **false** را برگردانید.

حالت پایه دیگر زمانی است که مقدار ستون N باشد، یعنی $j = N$ ، سپس به سطر بعدی بروید، یعنی $i++$ و $j = 0$.

اگر شاخص فعلی تخصیص داده نشد، عنصر را از ۱ تا ۹ پر کنی و برای هر ۹ حالت با شاخص عنصر بعدی، یعنی $i, j+1$ تکرار کنیم. اگر فراخوانی بازگشتی **true** را برگرداند، حلقه را شکسته و **true** را برگردانیم.

اگر شاخص فعلی اختصاص داده شود، تابع بازگشتی را با شاخص عنصر بعدی، یعنی $i, j+1$ فراخوانی کنیم.

پیچیدگی زمانی: $O(9^{(N*N)})$ ، برای هر شاخص اختصاص نیافته، ۹ گزینه ممکن وجود دارد، بنابراین

پیچیدگی زمانی $O(9^{(n*n)})$ است.

پیچیدگی فضایی: $O(N*N)$ ، برای ذخیره آرایه خروجی به یک ماتریس نیاز است.

Sudoku using Backtracking

مانند سایر مشکلات **Backtracking**، سودوکو را می توان با اختصاص اعداد یک به یک به سلول های خالی حل کرد. قبل از تخصیص شماره، بررسی کنیم که آیا اختصاص دادن آن ایمن است یا خیر.

بررسی کنیم که همان عدد در ردیف فعلی، ستون فعلی و زیرگروه فعلی 3×3 وجود نداشته باشد. پس از بررسی ایمنی، شماره را اختصاص دهید و به صورت بازگشتی بررسی کنید که آیا این تخصیص منجر به راه حل می شود یا خیر. اگر انتساب به راه حلی منجر نشد، عدد بعدی را برای سلول خالی فعلی امتحان کنید. و اگر هیچ یک از اعداد (۱ تا ۹) به راه حل منتهی نشد، **false** را برگردانید و هیچ جوابی را چاپ کنید.

برای حل مشکل مراحل زیر را دنبال کنید:

تابعی ایجاد کنید که بررسی می کند پس از اختصاص شاخص فعلی، شبکه ناامن می شود یا خیر. **hashmap** را برای سطر، ستون و کادر نگه دارید. اگر هر عددی دارای فرکانس بیشتر از ۱ در **hashmap** باشد، **false** را برگردانید، در غیر این صورت **true** را برگردانید. با استفاده از حلقه ها می توان از **hashMap** جلوگیری کرد. یک تابع بازگشتی ایجاد کنید که یک شبکه می گیرد.

مکان تعیین نشده را بررسی کنید.

در صورت وجود، عددی از ۱ تا ۹ را اختصاص می دهد.

بررسی کنید که آیا اختصاص شماره به شاخص فعلی، شبکه را ناامن می کند یا خیر.

اگر ایمن باشد، تابع را برای همه موارد ایمن از ۰ تا ۹ به صورت بازگشتی فراخوانی کنید.

اگر هر تماس بازگشتی درست را برگرداند، حلقه را پایان دهید و **true** را برگردانید. اگر هیچ تماس بازگشتی درست را برگرداند، **false** را برگردانید.

اگر مکان تعیین نشده ای وجود ندارد، مقدار واقعی را برگردانید.

پیچیدگی زمانی: $O(9^{(N*N)})$ ، برای هر شاخص اختصاص نیافته، ۹ گزینه ممکن وجود دارد، بنابراین پیچیدگی زمانی $O(9^{(n*n)})$ است. پیچیدگی زمانی یکسان باقی می ماند، اما کمی هرس اولیه وجود خواهد داشت، بنابراین زمان صرف شده بسیار کمتر از الگوریتم ساده لوح (اولیه) خواهد بود، اما پیچیدگی زمانی **upper bound** ثابت می ماند.

پیچیدگی فضایی: $O(N*N)$ ، برای ذخیره آرایه خروجی به یک ماتریس نیاز است.

- Worst Case Time Complexity: $O(9^m)$
- Average Case Time Complexity: $O(9^m)$
- Best Case Time Complexity: $O(m^2)$ [This takes place when the number of backtracking steps are minimized]

الگوریتم‌های Backtracking را می‌توان با افزودن روش‌های heuristics بهبود بخشید. می‌توانید تکنیک‌های فیلتر کردن و سفارش را اضافه کنید.

تفاوت بین brute-force و backtracking در این است که در backtracking، ما همه گزینه‌های ممکن را بررسی نمی‌کنیم، بلکه فقط گزینه‌های ارزشمند را بررسی می‌کنیم و راه حل را به صورت تدریجی می‌سازیم.

برای بهبود Backtracking :

با توجه به اسلاید ها از

General-purpose ideas give huge gains in speed

Ordering: - Which variable should be assigned next? –

In what order should its values be tried?

Filtering: Can we detect inevitable failure early?

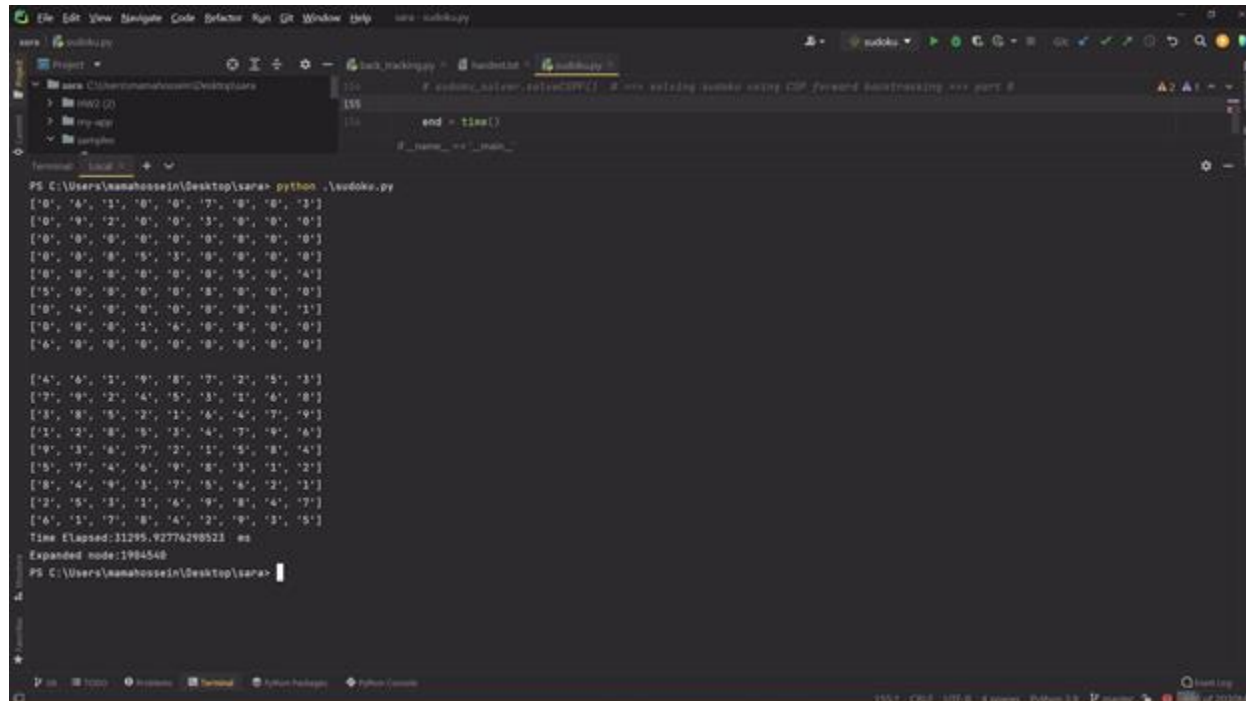
بررسی forward checking را زودتر از backtracking ساده تشخیص می‌دهد و بنابراین اجازه می‌دهد شاخه‌های درخت جستجو که منجر به شکست می‌شود زودتر از پس انداز ساده هرس شوند. این درخت جستجو و مقدار کلی کار انجام شده را کاهش می‌دهد.

الگوریتم Backtrack:

الگوریتم عقبگرد بسیار ساده است. این همان رویکردی است که در مسئله n-queen استفاده می‌شود. شرط اولیه ما این است که یک سلول خالی (که با '۰' نشان داده شده است) در جدول پیدا کنیم تا آن را با یک عدد پر کنیم. اگر نقطه خالی پیدا نکرد به این معنی است که جدول پر است و مشکل حل شده است. هر زمان که یک سلول خالی پیدا کند، بررسی می‌کند که کدام عدد در محدوده ۱ تا ۹ برای استفاده در سلول بی خطر است. پس از یافتن عدد مناسب، سلول را پر می‌کند و دوباره تابع backtracking را فراخوانی می‌کند تا

عمیق‌تر در درخت شیرجه بزند تا سلول بعدی پر شود. این فراخوانی تابع به صورت بازگشتی در هر مرحله انجام می‌شود 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. این هم‌کد:

همانطور که می‌بینیم تعداد زیادی گره را در درخت جستجو ردیابی می‌کنند. حالا بیا باید مشکل را بهینه کنیم و تعداد گره‌ها را کاهش دهیم.



```
PS C:\Users\mamahosseini\Desktop\sara> python .\sudoko.py
[[0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0]]

[[0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0]]

Time Elapsed: 11295.92776298521 ms
Expanded node: 1984548
PS C:\Users\mamahosseini\Desktop\sara>
```

اگر به خروجی نهایی کار دقت کنیم تعداد نودها و تایم پرینت گرفته شده که تعداد بسیار زیادی است و در مرحله بعد بهینه‌تر میشود!

```
File Edit Selection View Go Run Terminal Help
sudoku.py - Visual Studio Code

C:\Users\user> Desktop > Code_HW3_SaraYounesi > sudoku.py > Sudoku > isSafe

1 from time import time
2
3
4 class Sudoku:
5     def __init__(self, dim, fileBir):
6         self.dim = dim
7         self.expandedNodes = 0
8         with open(fileBir) as f:
9             content = f.readlines()
10            self.board = [list(x.strip()) for x in content]
11            self.rv = self.get_remaining_puzzle()
12
13    def isSafe(self, row, col, choice):
14        choiceStr = str(choice)
15        for i in range(self.dim):
16            if self.board[row][i] == choiceStr or self.board[i][col] == choiceStr:
17                return False
18
19        boxR = row - (row % 3)
20        boxV = col - (col % 3)
21        for i in range(3):
22            for j in range(3):
23                if self.board[boxR + i][boxV + j] == choiceStr:
24                    return False
25        return True
26
27    # for simple backtracking
28    def get_next_location(self):
29        for i in range(self.dim):
30            for j in range(self.dim):
31                if self.board[i][j] == '0':
32                    return i, j
33        return -1, -1
34
35    # this fun return valid choice per cell
36    def get_valid_domain(self, row, col):
37        valid_domain = [str(i) for i in range(1, self.dim + 1)]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

C:\Program Files\Python38\python.exe: can't open file 'sudoku.py': [Errno 2] No such file or directory
PS C:\Users\user>

Ln 13, Col 40 Spaces: 4 UTF-8 CRLF Python Select Interpreter Go Live Better ready Prettier

```
File Edit Selection View Go Run Terminal Help
sudoku.py - Visual Studio Code

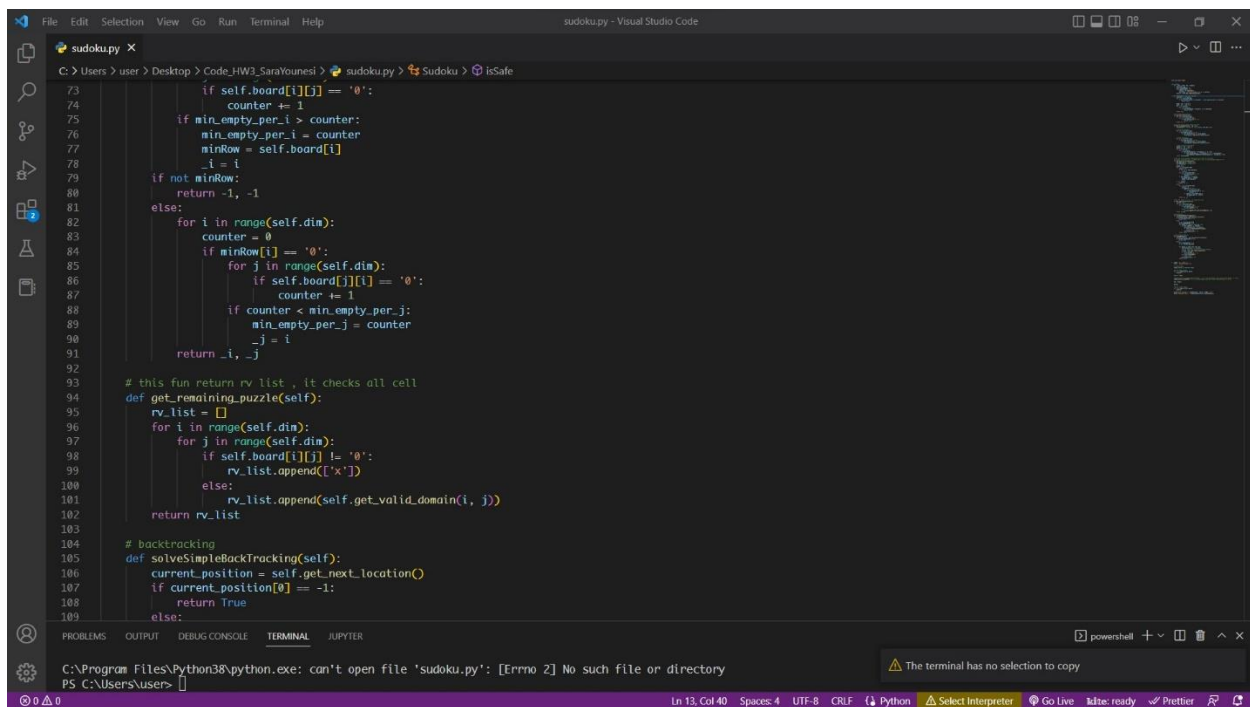
C:\Users\user> Desktop > Code_HW3_SaraYounesi > sudoku.py > Sudoku > isSafe

37 valid_domain = [str(i) for i in range(1, self.dim + 1)]
38
39 # check horizontally
40 for i in range(self.dim):
41     if self.board[row][i] != '0':
42         if self.board[row][i] in valid_domain:
43             valid_domain.remove(self.board[row][i])
44
45 # check vertically
46 for i in range(self.dim):
47     if self.board[i][col] != '0':
48         if self.board[i][col] in valid_domain:
49             valid_domain.remove(self.board[i][col])
50
51 # check index in its square
52 square_i = row - row % 3
53 square_j = col - col % 3
54 for i in range(3):
55     for j in range(3):
56         if self.board[square_i + i][square_j + j] != '0':
57             if self.board[square_i + i][square_j + j] in valid_domain:
58                 valid_domain.remove(self.board[square_i + i][square_j + j])
59 return valid_domain
60
61 # this fun return minimum remaining value index of self.board
62 # it means it prefer the i, j which are in row, col with minimum number of '0'
63 def get_next_mv_location(self):
64     min_empty_per_i = float('inf')
65     min_empty_per_j = float('inf')
66     _i, _j = 9, 9
67     minRow = []
68     for i in range(self.dim):
69         counter = 0
70         if '0' not in self.board[i]:
71             continue
72         for j in range(self.dim):
73             if self.board[i][j] == '0':
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

C:\Program Files\Python38\python.exe: can't open file 'sudoku.py': [Errno 2] No such file or directory
PS C:\Users\user>

Ln 13, Col 40 Spaces: 4 UTF-8 CRLF Python Select Interpreter Go Live Better ready Prettier



```
73         if self.board[i][j] == '0':
74             counter += 1
75         if min_empty_per_i > counter:
76             min_empty_per_i = counter
77             minRow = self.board[i]
78             i = j
79     if not minRow:
80         return -1, -1
81     else:
82         for i in range(self.dim):
83             counter = 0
84             if minRow[i] == '0':
85                 for j in range(self.dim):
86                     if self.board[j][i] == '0':
87                         counter += 1
88                     if counter < min_empty_per_j:
89                         min_empty_per_j = counter
90                         j = i
91         return i, j
92
93     # this fun return rv list , it checks all cell
94     def get_remaining_puzzle(self):
95         rv_list = []
96         for i in range(self.dim):
97             for j in range(self.dim):
98                 if self.board[i][j] != '0':
99                     rv_list.append(['x'])
100                 else:
101                     rv_list.append(self.get_valid_domain(i, j))
102         return rv_list
103
104     # backtracking
105     def solveSimpleBackTracking(self):
106         current_position = self.get_next_location()
107         if current_position[0] == -1:
108             return True
109         else:
```

C:\Program Files\Python38\python.exe: can't open file 'sudoku.py': [Errno 2] No such file or directory
PS C:\Users\user>

در این کد ها چند تابع پیاده سازی کردیم از جمله

```
def get_valid_domain
```

که بررسی می کند در سطر و ستون و باکس عدد تکراری نداشته باشیم .

برای هر قسمت از کد کامنت گذاشته شده است.

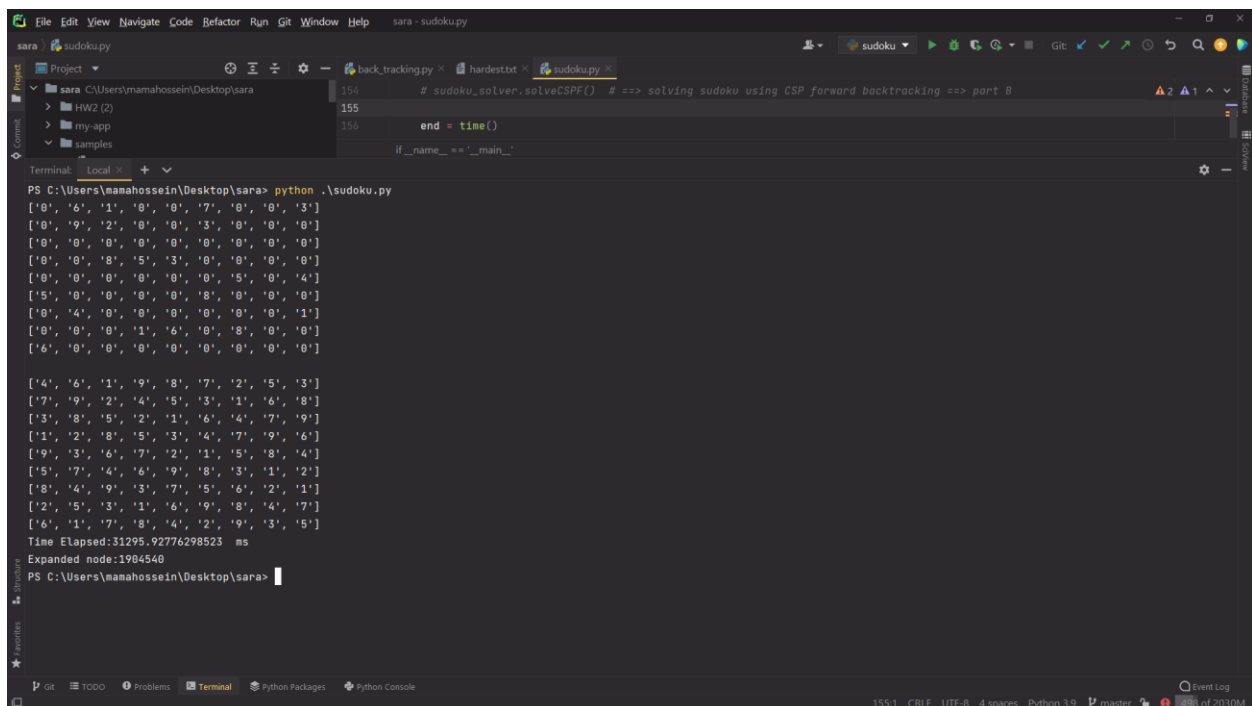
بررسی کلی الگوریتم CSP:

اگرچه الگوریتم های Backtracking تمایل دارند هر مشکلی را در تئوری حل کنند، اما به فضای زیادی از حافظه نیاز دارد و زمان زیادی را صرف می کند. الگوریتم های CSP به منظور کوچک کردن فضای بزرگ و تقویت الگوریتم ها معرفی شدند. با الگوریتم های Forward Checking خوب و توابع consistent heuristic ، حل مسئله با سرعت بالا با نیاز به حافظه کم امکان پذیر خواهد بود.

Backtracking یک تکنیک واقعا ساده است که بسیاری از مشکلات را حل می کند. با این حال، حل کننده های Backtracking تمایل دارند مشکلات را با جستجو در تمام فضای مشکل حل کنند و ممکن است همه موارد را برای یافتن راه حل بررسی کنند. آنها همچنین زمان زیادی را مصرف می کنند. هرچه مشکل پیچیده تر باشد، حل کننده کندتر می شود. اینجاست که الگوریتم های CSP پا به عرصه می گذارند تا این فضا را کوچک کرده و سرعت سیستم را افزایش دهند!

بخشی از الگوریتم های CSP الگوریتم های forward checking هستند. آنها برای بررسی سازگاری و انتشار محدودیت استفاده می شوند.

ما معمولاً اصطلاح CSP را در زمینه هوش مصنوعی می شنویم و انتظار داریم برخی ویژگی های هوشمند را در آن پیدا کنیم. با این حال، از نظر من الگوریتم های CSP فقط نسخه های بهینه شده رویکرد Backtracking هستند. در واقع، بیشتر شبیه تکنیکی است که برای سرعت بخشیدن به الگوریتم شما استفاده می شود. بیایید با یک مثال در این مورد بحث کنیم. مشکل معروف سودوکو! هدف ما کاهش تعداد گره های گسترش یافته در درخت جستجوی سودوکو تا حد امکان است. من یک کلاس حل سودوکو ایجاد کرده ام که قرار است برای ایجاد عامل ما استفاده شود. این کلاس شامل روش های حل کننده ای است که برای حل برد آن استفاده می شود.



```
sara - sudoku.py
sara \ sudoku.py
Project
sara C:\Users\mamahosseini\Desktop\sara
HW2 (2)
my-app
samples
Terminal
PS C:\Users\mamahosseini\Desktop\sara> python .\sudoku.py
['0', '6', '1', '0', '0', '7', '0', '0', '3']
['0', '9', '2', '0', '0', '3', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '0', '0', '0']
['0', '0', '8', '5', '3', '0', '0', '0', '0']
['0', '0', '0', '0', '0', '0', '5', '0', '4']
['5', '0', '0', '0', '0', '8', '0', '0', '0']
['0', '4', '0', '0', '0', '0', '0', '0', '1']
['0', '0', '0', '1', '6', '0', '8', '0', '0']
['6', '0', '0', '0', '0', '0', '0', '0', '0']

['4', '6', '1', '9', '8', '7', '2', '5', '3']
['7', '9', '2', '4', '5', '3', '1', '6', '8']
['3', '8', '5', '2', '1', '6', '4', '7', '9']
['1', '2', '8', '5', '3', '4', '7', '9', '6']
['9', '3', '6', '7', '2', '1', '5', '8', '4']
['5', '7', '4', '6', '9', '8', '3', '1', '2']
['8', '4', '9', '3', '7', '5', '6', '2', '1']
['2', '5', '3', '1', '6', '9', '8', '4', '7']
['6', '1', '7', '8', '4', '2', '9', '3', '5']

Time Elapsed: 31295.92776298523 ms
Expanded node: 1984548
PS C:\Users\mamahosseini\Desktop\sara>
```

الگوریتم CSP:

CSP مخفف Constraint Satisfaction Problem است. بنابراین، هدف اصلی ما برای طراحی چنین الگوریتمی برآورده کردن تمام محدودیت های تعریف شده ای است که مسئله معرفی می کند. 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 می دهد که کدام مقادیر را می توان به یک متغیر خاص اختصاص داد. و در نهایت، محدودیت ها نشان می دهد که کدام یک از مقادیر موجود در دامنه می تواند در لحظه مورد استفاده قرار گیرد. بیایید این تکنیک را روی مشکل سودوکو خود امتحان کنیم.

سه ویژگی مسئله به صورت زیر تعریف می شود:

متغیرها: هر سلول خالی روی تابلو

دامنه ها: برای هر سلول، یک دامنه به عنوان مجموعه ای از اعداد بین ۱ تا ۹ تعریف می شود، به جز اعدادی که قبلاً در ردیف، ستون یا مربع های 3×3 استفاده شده اند.

محدودیت ها: بدون اعداد اضافی در ردیف ها، ستون ها و مربع های 3×3 .

بنابراین اکنون طرح کلی الگوریتم CSP ما تعریف شده است و زمان آن رسیده است که بهینه سازی الگوریتم Backtrack را آغاز کنیم.

در مرحله اول، ما به آرایه ای از همه دامنه های همه متغیرها نیاز داریم. به عبارت دیگر، یک فاصله برای حفظ مقادیر باقیمانده برای هر متغیر مورد نیاز است. بنابراین، یک ویژگی به نام `rv` به کلاس ما اضافه می شود و در ادامه کد به عنوان `self.rv` بر اساس `python OOP` معرفی می شود. من تصمیم گرفتم دامنه مقادیر ثابت را روی برد با `['x']` جایگزین کنم فقط در صورتی که سلول شامل یک عدد ثابت باشد. در موارد دیگر، معیارهای سودوکو را بررسی می کنم تا مقادیر مناسب یک سلول را بیابم و آن را به لیست `self.rv` اضافه کنم:

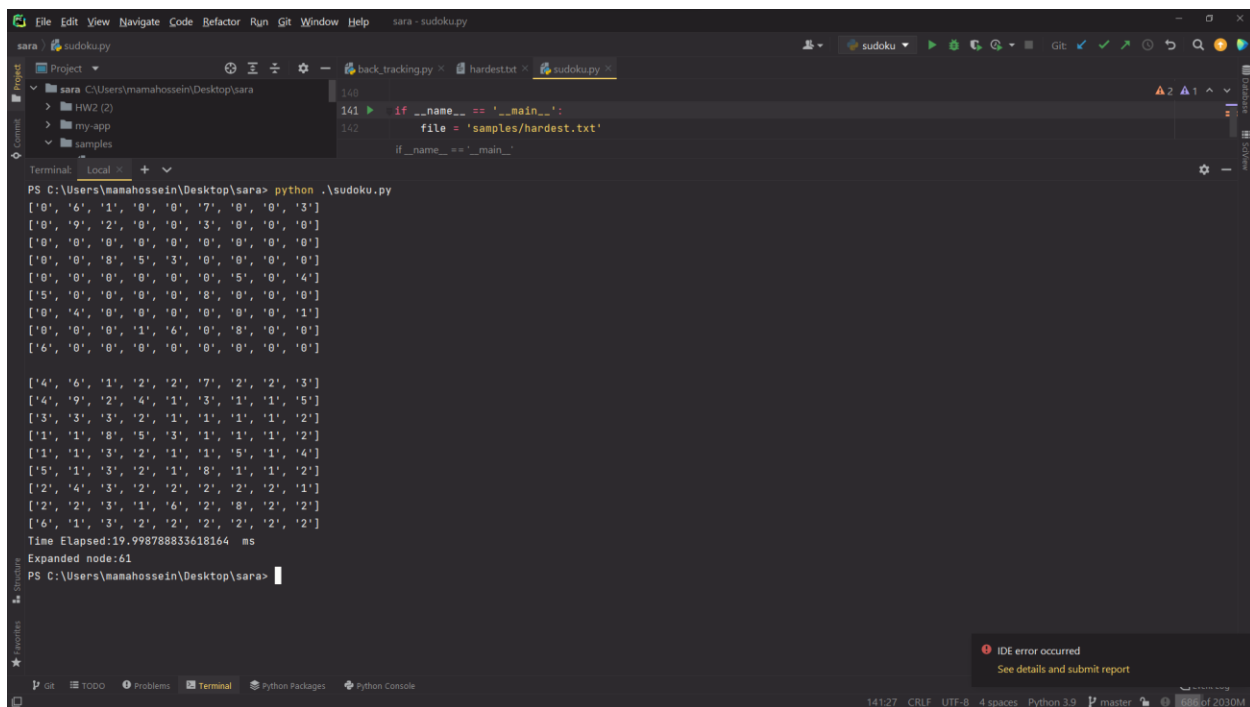
اکنون که اطلاعات آماده استفاده است، باید یک سلول خالی یا به عبارتی یک متغیر را به عنوان حرکت دوم انتخاب کنیم. اما آیا این مهم است؟ در واقع آن است که. برای این مشکل، ساده ترین روش این است که ابتدا سلول ها را با دامنه های کوچکتر پر کنید. به عنوان مثال، اگر دامنه یک سلول `[۳]` و دامنه سلول دیگر `[۱، ۲، ۹]` باشد، بدیهی است که پر کردن سلول با اندازه دامنه ۱ بهتر است زیرا این تنها انتخاب است و قطعاً درست است.

اگر این ایده را گسترش دهیم، اگر مقداری را از مجموعه دامنه کوچک انتخاب کنید، احتمال زیادی برای انتخاب مقدار مناسب خواهید داشت. بیایید تابع `getNextMRVRowCol()` را که `MRV` مخفف `Minimum Remaining Value` است، فراخوانی کنیم. همانطور که اشاره کردم، مقادیر ثابت را به عنوان `x` علامت گذاری کردیم، بنابراین ابتدا بررسی می کنیم که آیا یک سلول ثابت است و همچنین اگر دامنه خالی است، ۱۰ را به عنوان یک عدد بزرگ به خارج از فضای مشکل برمی گردانیم تا از انتخاب یک عامل خالی جلوگیری کنیم. دامنه به عنوان حداقل سلول مقدار باقی مانده.

`Forward Checking` صرفاً یک چشم انداز برای برنامه برای افزایش احتمال کسب سود از انتخاب خود در سطوح بالاتر درخت است. به عنوان مثال، تصور کنید که دو خانه `v1` و `v2` در یک ردیف با دامنه های `d1=[1,2]` و `d2=[1]` وجود دارد. تابع مکان ما `v1` را انتخاب می کند. در حالت عادی برنامه ابتدا ۱ را انتخاب می کند، در نتیجه مقدار جدید `d2=[2]` خواهد بود و در لایه عمیق تر متوجه می شود که هیچ مقدار ممکن برای سلول با دامنه `d2` وجود ندارد و به عقب برمی گردد.

اگر به خاطر داشته باشید هدف ما کاهش تعداد بسط گره ها بود که شامل حرکت عقبگرد است. به عنوان راه حل، ابتدا می توانیم با انتخاب ۱ بررسی کنیم، آیا انتخاب های احتمالی برای دیگری حذف می شود؟ خواهیم دید که تمام مقادیر باقی مانده برای `d2` را حذف می کند. در این مرحله به ما اطلاع داده می شود که انتخاب ۱ قطعاً در نتیجه یک حرکت عقبگرد هزینه دارد و بنابراین ما ۲ را به عنوان پاسخ ممکن انتخاب می کنیم.

امکان اجرای چک کردن رو به جلو برای بیش از یک مرحله وجود دارد. با این حال، ممکن است باعث سربار زمانی جدی شود. در این مورد، من ترجیح دادم روشی را پیاده سازی کنم که با انتخاب یک مقدار مشخص برای یک سلول بررسی کند که آیا فرصت های احتمالی برای سلول های دیگر موجود در برد را حذف می کند یا خیر. با جمع کردن همه اینها با هم.



```
sara - sudoku.py
Project
sara C:\Users\mamahosseini\Desktop\sara
HW2 (2)
my-app
samples
Terminal
Local
PS C:\Users\mamahosseini\Desktop\sara> python .\sudoku.py
[0, 6, 1, 0, 0, 7, 0, 0, 3]
[0, 9, 2, 0, 0, 3, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 8, 5, 3, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 5, 0, 4]
[5, 0, 0, 0, 0, 8, 0, 0, 0]
[0, 4, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 1, 6, 0, 8, 0, 0]
[6, 0, 0, 0, 0, 0, 0, 0, 0]

[4, 6, 1, 2, 2, 7, 2, 2, 3]
[4, 9, 2, 4, 1, 3, 1, 1, 5]
[3, 3, 3, 2, 1, 1, 1, 1, 2]
[1, 1, 8, 5, 3, 1, 1, 1, 2]
[1, 1, 3, 2, 1, 1, 5, 1, 4]
[5, 1, 3, 2, 1, 8, 1, 1, 2]
[2, 4, 3, 2, 2, 2, 2, 2, 1]
[2, 2, 3, 1, 6, 2, 8, 2, 2]
[6, 1, 3, 2, 2, 2, 2, 2, 2]

Time Elapsed:19.998788833618164 ms
Expanded node:61
PS C:\Users\mamahosseini\Desktop\sara>
```

همانگونه که در پرینت کد میبینیم تعداد expand node ها برابر ۶۱ شده است !

و تایم از ۳۱۲۹۵ به ۱۹,۹ کاهش یافته است !

Time /Space order ما به شدت کاهش داشته اند.

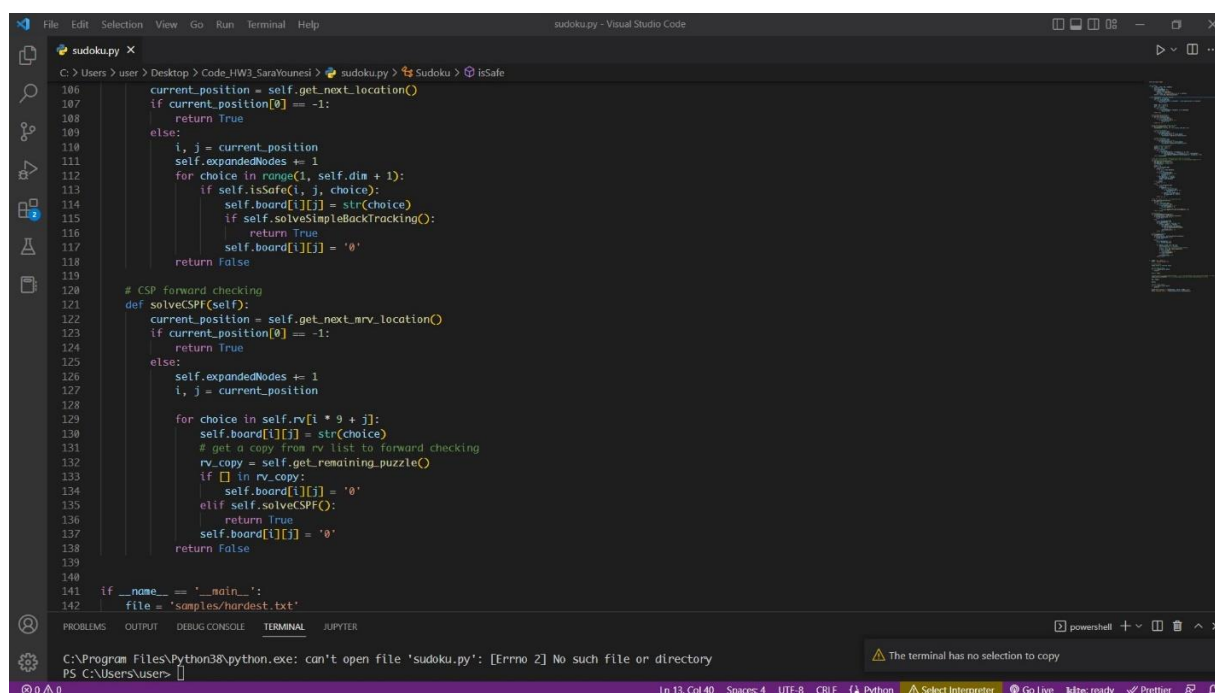
این واقعا فوق العاده است! ما فقط میزان گسترش گره را تقریباً ۸۵٪ کاهش دادیم. اگر متوجه شدید که در برخی موارد زمان سپری شده بیشتر از الگوریتم Backtracking است. این سرباری است که قبلاً به آن اشاره کردم. اگرچه فضای جستجوی خود را کاهش دادیم، اما به دلیل تحلیلی که روی مشکل اعمال کردیم، زمان جستجو افزایش یافت. روش بررسی رو به جلو که ما استفاده کردیم نمونه ای از روشی با سربار زمانی بود. اگرچه در مسائل کوچک هیچ تفاوتی نمی کند و در برخی موارد مانند مثال حتی بسیار سریعتر عمل می کند اما در مسائل پیچیده متفاوت است. راه بهتر این است که فقط دامنه مقادیر موجود در همان سطر و ستون سلول انتخاب شده را بررسی کنید نه کل جدول را!

علاوه بر این، می توانید توابع اکتشافی را به منظور حدس زدن اینکه کدام مقدار در یک دامنه متغیر خاص انتخاب شود بهتر است پیاده سازی کنید. به عنوان مثال، تابع اکتشافی شما می تواند مقداری را انتخاب کند که کمترین مقدار در جدول تکرار شده است. اگر روش شما ثابت باشد، قطعاً نتایج شما را بهبود می بخشد.

الگوریتم های جستجوی CSP در بدترین حالت نمایی هستند. یک upper bound بی اهمیت در پیچیدگی زمانی الگوریتم های جستجوی CSP $O(d^n)$ است، که در آن n و d به ترتیب تعداد متغیرها و حداکثر اندازه دامنه CSP زیربنایی هستند.

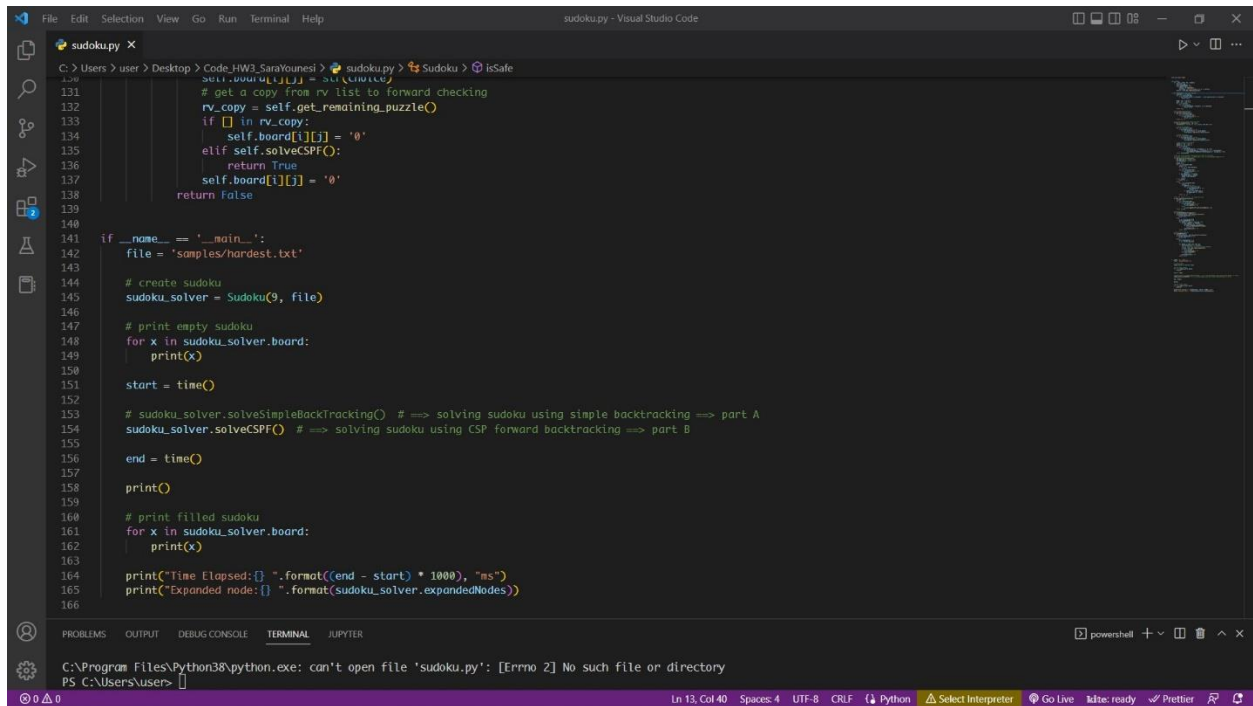
ترکیبی از روش های اکتشافی حل محدودیت می تواند پیچیدگی زمانی را کاهش دهد. به طور خاص، ما ثابت می کنیم که الگوریتم FC-CBJ همراه با اکتشافی مرتب سازی متغیر اول شکست (FF) به پیچیدگی زمانی $O((d-1)n)$ می رسد، که در آن n و d تعداد متغیرها و حداکثر هستند. اندازه دامنه CSP داده شده، به ترتیب. علاوه بر این، نشان می دهیم که ترکیب ضروری است زیرا نه FC-CBJ به تنهایی و نه FC با FF به پیچیدگی فوق نمی رسند. نتایج پیشنهادی جالب هستند زیرا آنها ارتباطی بین رویکردهای نظری و عملی برای تحقیقات CSP برقرار می کنند.

ما در کل در این مسائل از Backtrack و CSP forward consistant پیاده سازی کردیم و میشود برای انتخاب لوکیشن بعدی کدی پیاده سازی کنیم که برای آن بهینه تر و مفید تر باشد. کد های قسمت CSP در ادامه همراه کامنت های مربوطه آورده شده است.



```
106 current_position = self.get_next_location()
107 if current_position[0] == -1:
108     return True
109 else:
110     i, j = current_position
111     self.expandedNodes += 1
112     for choice in range(1, self.dlm + 1):
113         if self.isSafe(i, j, choice):
114             self.board[i][j] = str(choice)
115             if self.solveSimpleBacktracking():
116                 return True
117             self.board[i][j] = '0'
118     return False
119
120 # CSP forward checking
121 def solveCSP(self):
122     current_position = self.get_next_mrv_location()
123     if current_position[0] == -1:
124         return True
125     else:
126         self.expandedNodes += 1
127         i, j = current_position
128
129         for choice in self.rv[i * 9 + j]:
130             self.board[i][j] = str(choice)
131             # get a copy from rv list to forward checking
132             rv_copy = self.get_remaining_puzzle()
133             if choice in rv_copy:
134                 self.board[i][j] = '0'
135             elif self.solveCSP():
136                 return True
137             self.board[i][j] = '0'
138         return False
139
140 if __name__ == '__main__':
141     file = 'samples/hardest.txt'
```

C:\Program Files\Python38\python.exe: can't open file 'sudoku.py': [Errno 2] No such file or directory
PS C:\Users\user>



```
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166

self.board[i][j] = self.choice()
# get a copy from rv list to forward checking
rv_copy = self.get_remaining_puzzle()
if [] in rv_copy:
    self.board[i][j] = '0'
elif self.solveCSPF():
    return True
self.board[i][j] = '0'
return False

if __name__ == '__main__':
    file = 'samples/hardest.txt'
    # create sudoku
    sudoku_solver = Sudoku(9, file)
    # print empty sudoku
    for x in sudoku_solver.board:
        print(x)
    start = time()
    # sudoku_solver.solveSimpleBacktracking() # ==> solving sudoku using simple backtracking ==> part A
    sudoku_solver.solveCSPF() # ==> solving sudoku using CSP forward backtracking ==> part B
    end = time()
    print()
    # print filled sudoku
    for x in sudoku_solver.board:
        print(x)
    print("Time Elapsed: {}".format((end - start) * 1000), "ms")
    print("Expanded node: {}".format(sudoku_solver.expandedNodes))

C:\Program Files\Python38\python.exe: can't open file 'sudoku.py': [Errno 2] No such file or directory
PS C:\Users\user>
```