**فصل 1:**

هوش مصنوعی 4 رویکرد دارد: تفکر عاقلانه، عمل عاقلانه، تفکر انسان گونه و عمل انسان گونه، انسان چون هوشمند هست و کار هوشمند انجام میدهد پس میتوانیم به عنوان معیار ازش استفاده کنیم و عاقلانه هم یک معیار دیگر میتواند باشد یعنی اون معیار همیشه کار درست را انجام میدهد با توجه به دانش فعلی عاقلانه همان معنی بهینه هم میدهد.

رویکرد عمل انسان گونه: تورینگ اولین بار واسه این یک تست نوشته، به این صورت هست که اگر سیستمی بتواند به نوعی جواب بدهد که 30 درصد از جواب هایش را انسان نفهمد که این مال انسان هست یا اون هوشمندی که میتوانیم بگوییم انسان گونه رفتار کرده هست. حالا این سیستم برای رسیدن به این میزان چه ویژگی های داشته باشد؟ اولیش پردازش زبان طبیعی برای برقراری ارتباط هست دوم بازنمایی دانش برای ذخیره هست و همچنین بر روی این دانش ها باید بتواند استدلال انجام بدهد و استدلال خودکار باشد و چهارمین ویژگی این هست که سیستم دارای یادگیری باشد یعنی صرفا یک سری چیز را تکرار نکند بلکه با شرایط جدید هماهنگ شود و الگو های جدید کشف کند. این 4 تا ویژگی برای پاس کردن تست تورینگ هست.   
تست تورینگ کامل: 2 ویژگی به ویژگی های بالا اضافه میشود که اولیش روباتیک و دومیش قدرت بینایی ماشین هست و همه اینا یک شاخه از علم هوش مصنوعی هستند.

رویکرد تفکر انسان گونه: باید یاد بگیریم که انسان چه گونه فکر میکند 3 روش وجود دارد درون گرایی : واسه وقتی هست که انسان داره فکر میکنه، آزمایش های روان شناسی و عکس برداری از مغز.

رویکرد عمل انسان گونه باعث شکل گرفتن علوم جدید شد که مهم ترین آن علوم شناختی هست.

رویکرد تفکر عاقلانه: این سیستم ها درست تفکر و استدلال میکنند، با یک سری مکانیزم میتوانستند تمام زبان طبیعی را به شکل منطق خودشان در بیاورند. ولی چون محدودیت زمانی و غیر ممکن بودن پیاده سازی هم داریم واسه همین جلوی این موضوع را گرفته ایم و همچنین دانش منطقی اطلاعات دقیق هست ولی ما در اینجا داده های احتمالی داریم و باید با نظریه احتمال کار کنیم و کلا این رویکرد از لحاظ زمان و منابع برای ما خوب نیست و صرفا روی استدلال تمرکز دارد و عمل نمیکند.

رویکرد عملکرد عاقلانه: در این رویکرد موجودیت هایی میسازیم که میخواهیم عمل کنند که به این میگوییم عامل، عامل هر چیزی هست که در محیط عمل کند و بتواند محیط را درک کند. و به دنبال اهداف هست و تغییر پذیر هست.

یعنی عامل در هر لحظه تصمیمی را میگیرد که بهترین خروجی مورد انتظار را دارد یا منجر به بهترین نتیجه با توجه به دانشی که دارد چون محیط ما غیر قطعی هست. چرا این رویکرد رو ترجیح میدهیم و کتاب هم این رویکرد را در نظر دارد؟ مثلا نسبت به سوم فقط استدلال و تفکر میکند و ما باید عمل کنیم. مثل دست زدن به یک چیز داغ خوب اینجا سریع عمل کنیم بهتر هست یا استدلال کنیم و نتیجه گیری کنیم پس رویکرد سوم زیر مجموعه رویکرد چهارم هست. برتری که نسبت به انسان دارد بخاطر این هست که بهینه بودن یک حالت ریاضی هست و ما میتوانیم به عنوان معیار قرار بدهیم ولی وقتی انسان را قرار میدهیم از لحاظ تحقیقاتی سخت میشود چون رفتار ها متفاوت هست و همه رفتار ها بدرد ما نمیخورد و نمیتوانیم علم را جلو ببریم. Perfect rationality همیشه ساده نیست مخصوصا در محیط های پیچیده پس ما معمولا خود را محدود میکنیم به limited rationality. مثل بازی شطرنج در فصل بازی ها. اگر اون 4 تا ویژگی را که در تست تورینگ گفتیم عامل ما داشته باشد میتواند با محیط تعامل داشته باشد و تصمیم درست را بگیرد این برای تفکر انسان گونه. ما میتوانیم یک عامل را بدون اینکه نیاز باشد که ویژگی های تست تورینگ را داشته باشد و صرفا بر اساس تصمیم عقلانی بسازیم و بر اساس اون معیار عقلی که دارد تصمیم گیری کند و نیازی ندارد بداند انسان در آن لحظه چه کاری میکند.

**پایان جزوه اول و ویدیو اول.**

شروع ویدیو دوم:

هر موجودیتی که یک سری سنسور برای درک محیط دارد و میتواند یک سری action در محیط به کمک عملگرهاش انجام بدهد مثل انسان یا برنامه نرم افزاری عامل نام دارد.

به ادراکی که عامل از محیط در هر لحظه خاص دریافت میکند percept میگوییم. و تاریخچه همه این ادراک در زمان های مختلف دنباله ادراک گفته میشود. انتخاب های عامل به دانش قبلی و دنباله ادراک بستگی دارد. رفتار یک عامل به آینده بستگی ندارد. اگر تابعی به نام عامل داشته باشیم که به ازای دنباله ای از ادراک چه خروجی یا چه action انجام میدهد تابع عامل میگوییم و مثل یک تابع انتزاعی رضایی هست و ورودی آن دنباله ی ادراک هست و خروجی آن action هست. برنامه عامل برنامه ای هست که تابع عامل را روی سخت افزار پیاده سازی میکند. همون کد هستش. و یک مشکلی دارد چون دنباله ها خیلی زیاد هستند و شاید بینهایت شود واسه همین پیاده سازی آن مشکل دارد.

عامل عقلایی: rational بودن یعنی بهترین کار بهترین کار حالا یعنی چی؟ دنباله از اکشن ها باعث تغییر در محیط میشود و محیط به یک حالت دیگری میرود و وقتی محیط به حالتی برود که برای ما خوب هست میگوییم عامل خوب عمل کار کرده پس ما باید یک معیار کارایی تعریف کنیم.

تا نسبت به محیط ما خوب بودن را بسنجیم نه نسبت به رفتار ها و اکشن های یک عامل. مثلا در جارو برقی مقدار خاکی که عامل تمیز میکند نمیتواند باشد چون هعی جمع میکند و اون ها را که جمع کرده بیرون میریزد تا بینهایت شرط خودش را برآورده کند این رفتار عامل هست بلکه باید روی محیط تمرکز کنیم. تمیز بودن اتاق در هر واحد زمانی معیار بهتری هست.

برای بررسی بهترین عمل عامل 4 تا معیار داریم: معیار کارایی، دانش قبلی از محیط: چون اکشن به دانش قبلی بستگی دارد چون نسبت به شناخت عمل میکند دیگه، اعمال قابل انجام برای عامل: یعنی عامل را بر اساس اکشن ها باید مقایسه کنیم پس حتما باید بدونیم چه اعمالی میتواند عامل انجام دهد، معیار آخر دنباله ادراکی عامل تا این لحظه هست که بررسی کنیم سنسور هایی که یک عامل دارد چه قدر قوی هستند تا ببینیم عامل چه قدر عقلایی عمل میکند، مثل جارو برقی که همزمان میتواند هر دو اتاق را بررسی کند با جارو برقی که فقط یک اتاق را میتواند بررسی کند این مثال برای دو معیار آخر هست.

پس اول اون 4 تا معیار را برای هر مسئله ای تعریف کن. بعد مثلا چک کنیم عامل ها تو کمترین زمان مثلا انجام بدهند. یعنی اول اون 4 تا را مشخص کن بعد بر اساس اون عقلایی را تصمیم میگیریم.

دانای کل خروجی واقعی عمل خود را از قبل میداند و طبق آن عمل میکند ولی عامل عقلایی معیار کارایی مورد انتظارش را طبق ادراک تا حالا ماکسیمم میکند یعنی نهایتش بر اساس دانش قبلی هست. و اگر کار اشتباه کنیم نشانه بد بودن نیست چون ما فقط خواستیم معیار کارایی تا حالا را ماکسیمم میکند. جمع آوری اطلاعات یکی از مهم ترین بخش ها واسه عمل عقلایی هست که انجام کارهایی که ادراک را در آینده را تغییر میدهند.

امید ریاضی یک متغیر تصادفی: احتمال ضربدر اون متغیر تابع.

یک عامل بتواند علاوه بر جمع آوری اطلاعات باید قابلیت یادگیری هم داشته باشد، معمولا به این شکل هست که یک سری دانش اولیه و جغرافیایی اولیه را میدهیم و بعدش مثلا عامل توزیع خاک را پیدا کند این میشود خودمختاری یا یادگیری و عامل خودش را اصلاح میکند. پس یادگیری هم مهم هست. اگر به عامل هیچی نگیم و فقط خودش یاد بگیر عامل خود مختار هست. پس یک عامل عقلایی باید خود مختار باشد.

محیط وظیفه: برای حل مسئله باید 4 مورد را بررسی کنیم اکشن ها، سنسور ها، محیط و معیار کارایی را تعیین کنیم. محیط همان محیط وظیفه هست و مجموع همین 4 تا را در نظر بگیر.

محیط فیزیکی یک بخشی از محیط وظیفه هست.

خواص محیط اجرا: این یکی از اون 4 تای بالاست، اولیش محیط کاملا مشاهده پذیر یعنی به حالت کامل یک محیط دسترسی دارند در مقابل پاره ای مشاهده پذیر هست مثلا جارو برقی در هر لحظه هر دو اتاق را ببیند پس این محیط کاملا مشاهده پذیر هست. اگر عامل هیچ سنسور نداشت محیط غیر قابل مشاهده پذیر هست. موثر کاملا مشاهده پذیر: اون قسمتایی که نمیبینی باهاشون کاری هم نداری.

خواص محیط اجرا: محیط میتواند تک عامله مثل شطرنج و چند عامله مثل شطرنج باشد سوال این هست که عامل دیگری رو چطور تشخیص بدیم؟ مثلا سایر راننده تاکسی ها؟ ابجکت ب برای عامل الف یک عامل محسوب میشود اگر رفتار ب مبتنی بر ماکسیمم کردن معیار کارایی شود که مقدار آن بر رفتار الف تاثیر بگذارد. در برخی محیط رقابتی رفتار تصادفی برای غیر قابل پیش بینی بودن موثر هست.

خواص محیط اجرا: قطعی بودن یا تصادفی بودن، اگر حالت بعدی محیط کاملا به وسیله حالت فعلی و اکشن انتخاب شده عامل تعیین شود محیط قطعی هست. در واقع نتیجه عمل یک حالت یکتا هست.

یک محیط نامشخص است اگر قطعی یا کاملا مشاهده پذیر نباشد.

خواص محیط: اپیزودیک در مقابل ترتیبی: اپیزودیک یعنی اکشن هایی که بعدا انجام میدهیم و دنباله ادراکی که دریافت میکنیم هیچ ربطی به اکشن های انجام شده قبلی ندارد در واقع عامل نیازی به تفکر به اکشن های آینده ندارد، مثلا چک کردن کالا ها تو یک خط تولید، یا ربات هایی که عمل دسته بندی انجام میدهند مثلا عکس های ماهواره ای را دسته بندی کنند. برای محیط ترتیبی مثل همون بازی شطرنج یعنی در هر لحظه تصمیمی که میگیره بستگی دارد تو چه حالتی هستیم و چه اکشن هایی را انجام دادیم و قرار است انجام بدهیم، محیط های اپیزودیک برای ما بهتر هست.

محیط ایستا در مقابل پویا: در محیط ایستا حین تصمیم گیری عامل تغییر نمیکند و محیط پویا در حین تصمیم گیری عامل محیط تغییر میکند. مثل مثال تاکسی، یک حالت دیگری هم داریم که شبه پویا هست یعنی محیط ثابت بماند اما معیار کارایی تغییر میکند مثل شطرنج با ساعت.

محیط گسسته در مقابل پیوسته: هم با توجه به محیط، باید ببینیم ادراک و اکشن های ما و هندل شدن زمان و کمیت های ادراک و اکشن باید تعیین کنیم که محیط گسسته هست یا پیوسته مثلا شطرنج یک محیط گسسته هست اگر زمان هم بود باید گسسته باشد اکشن ها هم گسسته هستند یعنی قابل اندازه گیری و شمارش و محدود هستند ولی مثلا مثال تاکسی پیوسته هست و محیط های پیوسته پیچیده تر هستند نسبت به محیط های گسسته و همچین محیط پویا پیچیده تر هست نسبت به محیط ایستا.

اگر عامل قوانین محیط را بدانید محیط برایش شناخته شده هست یعنی هر اکشنی که انجام میدهد به چه نتیجه ای منجر میشود. حالا فرق با قطعی و غیر قطعی بودن چی هست؟ در قطعی و غیر قطعی ما نتیجه اکشن ها را میدانیم یعنی میدانیم حدودی چه نتیجه ای دارد حالا این میتواند یک نتیجه باشد یا دو یا چند نتیجه که غیر قطعی هست ولی در محیط ناشناخته ما همین را هم نمیدانیم. مثلا ما بازی تخته نرد را بلدیم و چون تاس داریم محیط غیر قطعی هست. یکی بازی شناخته شده میتواند کامل مشاهده پذیر نباشد و ناشناخته میتواند باشد ولی کامل مشاهده پذیر باشد. بحث شناخته شده ویژگی محیط نیست به عامل ربط دارد و دقت کن این خواص محیط هیچ تاثیری رو همدیگه نمیگذارند یعنی محیط پویا میتواند مشاهده پذیر یا قسمتی مشاهده پذیر یا غیر قابل مشاهده پذیر باشد.

در این درس کل تمرکز ما روی برنامه عامل هست و وارد قسمت معماری نمیشویم، یک ویژگی تمام برنامه عامل این هست که ورودی آن ادراک فعلی هست. ورودی تابع عمل یک چیز انتزاعی هست نه ادراک، اون برنامه چطوری مثلا عمل میکند؟ فرض کن تا اون دو تا A تمیز رفتی و الان A dirty به عنوان percept میاد میریم اینو میسچبونیم به بقیه و میگردیم تو جدول پیداش میکنم و قسمت راست را نگاه میکنم و اکشن را برمیداریم.

اولین نوع برنامه عامل: عامل انعکاسی ساده هست که تصمیم را فقط بر اساس ادراک فعلی تصمیم میگیرد و به همین دلیل عامل ساده هست، ولی فقط وقتی میتوانند درست تصمیم بگیرند که محیط کاملا مشاهده پذیر باشد و نیازی به دنباله ادراک نباشد و ادراک فعلی کافی هست و بفهمیم حالت فعلی جهان چطوری هست، که تو اکثر موارد این رو ما نداریم.

عامل انعکاسی مبتنی بر مدل: واسه وقتی هست که محیط پاره ای مشاهده پذیر هست و اون قسمتی که نمیبینیم را مدل میکنیم و ما یک حالت درونی داریم که اون قسمت از جهان که نمیبینیم را حدس میزند. و طبق مثال تاکسی که موقعیت ماشین تغییر میکند باید بر اساس اکشن های خودمون و قوانین جهان بیایم حالت درونی را اپدیت کنیم و به طور دقیق نیست البته.

عامل مبتنی بر هدف: خیلی وقتا هست که تصمیم درست بر اساس حالت فعلی و اکشن های خودمون بدست نمیاد اما تصمیم درست علاوه بر اینها به مقصد هم بستگی دارد.

پس باید به هدف نگاه کنیم و اطلاعات هدف را با مدل حالت قبلی ترکیب کند و بر اساس این تصمیم بگیرد، سرعت تصمیم گیری ما کمتر هست ولی منعطف هست یعنی اگر هدف یا مقصد تغییر کند عامل های مبتنی بر هدف میتوانند سریعتر خود را اصلاح کنند.

عامل مبتنی بر سودمندی : درسته دنبال این هستیم به هدف برسیم اگر به هدف رسیدیم خوشحالیم بقیه حالت ها برامون مهم نیست ولی توی این عامل مبتنی بر سودمندی میاد به همه مسیر ها یک عدد نسبت میدهد به عنوان تابع سودمندی، و با توجه به اینها مشخص میشود که کدوم مسیر بهتر هست. فرق معیار کارایی با سودمندی چیست؟ سودمندی یک چیز درونی عامل هست ولی کارایی نسبت به محیط تعریف میشود، یعنی در محیط مقدار معیار کارایی فرقی ندارد ولی تابع سودمندی میتواند متفاوت باشد و اگر تابع سودمندی منطبق باشد با معیار کارایی، یک عامل عقلایی در نظر گرفته میشود. یعنی اگه این رو بیشینه کردی اونم بیشینه شود، لزومی ندارد حتما تابع سودمندی را داشته باشیم اما بسیار منعطف هستند و قابلیت یادگیری دارند در شرایطی که اهداف متناقض هست مثل مثال تاکسی که هم سالم برسیم هم به موقع و ... ، و همچنین شرایطی کی اهداف غیر قطعی وجود دارند و به کمک تابع سودمندی انتخاب میکنیم یعنی بین اون اهداف با در نظر گرفتن تابع سودمندی و احتمالات مختلف میتوانیم بهترین هدف را برداریم. برای ماکسیمم کردن سودمندی مورد انتظار که میشود مقدار تابع ضربدر احتمال.

حالا چطوری به اون 4 تا عامل، یادگیری هم اضافه کنیم؟ از یک نقاد کمک میگیرد به عنوان بازخورد از محیط و بر اساس این نظرات میاد بهبود میدهد و نقاد هم با استفاده از استاندارد کارایی و سنسور ها که از محیط برداشت میکند چک میکند که عامل دارد چه قدر درست کار میکند، مولد مسئله هم باعث اکتشافات جدید میشود و باعث میشود دانش ما بروز شود یعنی اکشن های جدید را پیشنهاد میکند، مثلا اگه یهو بپیچی چپ چون راننده ها فحش میدن پس کار بدی هست انجام ندیم استاندارد میشود فحش دادن بقیه و مولد مسئله هم میشود اینکه بریم جای جدید رانندگی کنیم چیز جدید یاد بگیریم.

جلسه سوم:

اگر ما هدفی داشتیم که تو همون لحظه به آن نمی رسیدیم نیاز هست که یک برنامه ریزی یا planning انجام بدهیم، در واقع عامل حل مسئله با جستجو و بین راه های موجود میاد برنامه ریزی میکند و بین اونها یکی را انتخاب میکند.

اولین مرحله این هست که هدف را تعیین کنیم تا بفهمیم چه اکشن هایی باید انجام بدهیم، حالت یعنی میشود بودن در یک شهر. اکشن هم یعنی رفتن از یک شهر به شهر دیگر. در بخش سوم باید جستجو کنیم عامل میاد یک برنامه را بین برنامه های موجود جستجو میکند برای رسیدن به هدف و این قسمت باید الگوریتم ها را استفاده کنیم. مرحله آخر هم اجرا هست. اگر محیط مشاهده پذیر کامل و قطعی باشد عامل میتواند پرسپت ها را نادیده بگیرد یعنی وقتی رسید به کرج مطمئن هست تو کرج هست.

تست هدف یعنی یک حالت ورودی آیا هدف هست حالت هدف هست یا خیر.

ترنزیشن: نتیجه هر اکشن هست خروجی یعنی ما به چه حالتی میرسیم .

برای انتخاب حالت باید هدف تعیین کنی اگر هدف بودن در یک شهر هست پس حالت در یک شهر هست. در واقع حالت باید جوری باشد که قابل مقایسه با هدف باشد. و اون توابع اکشن و result را باید بتوانی بدست بیاری. یال ها اکشن ها هستند. تا جایی که ادامه میدهیم که دیگه همه چی تکرار میباشد و دیگر حالت مجزا نداریم. در مسئله هشت پازل هر حالت این خانه ها یک حالت هستند و باید به حالت هدف برسند. به الگوریتم جستجو فقط حالت اولیه را بهش میدهیم و تابع اکشن و ریزالت را خودش دارد و خود اون الگوریتم یک درخت جستجو میسازد، هر حالت را به یک گره یا نود تبدیل میکنند. به این داده ساختار در کامپیوتر میگوییم گره یا نود. در جستجو اول تست هدف میکنیم اگر هدف نبود اکشن اعمال میکنیم و به درخت اون نتایج جستجو را اضافه میکنیم در واقع درخت حاصلی از اکشن ها و ریزالت ها هستند. در هر لحظه گره هایی که تولید شده اند ولی بسط داده نشده اند در frontier قرار میگیرند و از اون برمیداریم و گسترش میدهیم پس یک داده ساختار هست.

توی تابع expand اول میاد حالت اون نود را میگیرد و بعد اکشن انجام میدهد و بعد تبدیل به نود میکند و اضافه میکند. با ریزالت حالت جدید میسازیم.

الگوریتم های درختی به حالت های دیده شده کاری ندارند یعنی تکراری هم میتواند باشد.

ورژن گرافی یعنی از حالت های تکراری صرف نظر میکند. با یک مجموعه به نام explored. اگر ما یک گره ای داشتیم که در فرانتیر بود و هنوز اکسپند نشده بود اون ورژنی را نگه میداریم که هزینه کمتری داشته باشد و اون هزینه بیشتر را پاک میکنیم و هزینه کمتر را بسط میدهیم . مسیر زاید: هزینه بیشتری دارد.

ضریب انشعاب: تعداد فرزندان یک گره مثلا میتوانی بگی حداکثر اکشن هایی که میتواند انجام دهد.

ادامه فصل سوم:

تمام تفاوت الگوریتم های سرچ، اولویت برداشتن از صف فرانتیر هست. الگوریتم های نا اگاهانه یعنی اینکه جز همین اطلاعات موجود مسئله هیچ اطلاعات دیگه ندارند یعنی حالت اولیه تابع اکشن که تو هر حالت چه عملیاتی میتوانیم انجام بدهیم، تابع ریزالت که همون حاصل اکشن هست تست هدف و هزینه مسیر و توصیف هر حالت مسئله اینها را دارند.

مسائل اکتشافی یک سری اطلاعات وابسته به مسئله هم دارد که با آن میتوانیم الگوریتم را بهبود دهیم.

BFS: کمترین عمق هست. اون تابع f میشود عمق گره. استثنا میشود تست هدف در زمان تولید گره صورت بگیرد و الگوریتم سریع تر میشود. تنها وقتی بهینه هست که هزینه همه گره ها یکسان باشد یعنی هدف با کمترین عمق پیدا کند. چرا نسخه گرافی؟ چون تکراری ها را بسط نمیدهد و از فرانتیر بزرگتر نمیشود. مشکل آن نمایی بودن پیچیدگی زمانی هست.

DFS: عمق بیشتر توی اونا چپ تری. چرا منفی عمق گره F باشد؟ چون عمق بیشتر شود. از نسخه درختی استفاده میشود. بزرگترین مشکل دور هست. اگر گرافی میرفتی این مشکل نداشت. مزیت آن پیچیدگی حافظه هست. حافظه آن BM هست چون نهایتا M سطح داریم که تو هر کدوم به اندازه ضریب انشعاب گره وجود دارد. این بزرگترین مزیت درختی هست تازه بعد از پیمایش هم از مجموعه پاک میشود. ولی اگر گرافی باشد اندازه explore set میشود اندازه فضای حالت که خیلی بزرگ هست یا نمایی هست. الگوریتم backtracking عین این الگوریتم هست فقط به جای اینکه گره های خواهر و برادر را نگه دارد فقط گره ای که بسط میخواهد بده را نگه میدارد. پیچیدگی زمانیش بدتر است چون اولین عمق تا عمق جواب میرفت این تا عمق مسئله میرود.

UCS: بر اساس هزینه مسیر گره ها کار میکند و کمترین مسیر را برمیدارد. الگوریتم گرافی هست اگر جهت ندادن دو طرفه هست. یادت باشد در گرافی گره ای میماند که هزینه کمتری با بهینه تری دارد. نکته ای که هست این هست که از شروع S تا گره N تمام گره های بهینه انتخاب شده اند و مسیر فعلی بهترین مسیر ممکن هست و بین این دو تا هیچ گره ای نیست که انتخاب نشده باشد . و حتما گسترش داده شده اند. هزینه اکشن ها باید مثبت باشد یعنی افزایشی و هزینه ها مثبت باشد. پس ما به هر هدفی میرویم قبلا بهترین مسیر را پیدا کرده ایم. کامل هست به این شرط که یک کف مثبت مثلا یک میلیون ام را براش در نظر بگیریم تا کامل باشد. این الگوریتم همه گره های اطراف را به ترتیب هزینه مصرف چک میکند و هر کی کمتر بود زودتر چک میشود تا به هدف برسیم. C\* هزینه بهینه مسیر تا هدف. این میتواند بیشتر از b به توان d باشد و خیلی کند تر باشد. اگر هزینه یال ها ثابت باشد انوقت c\* روی اپسیلون میشود خود d.

DLS: L خوب باید باشد. کامل هست به شرطی که جواب داخل اون محدوده L باشد. بهینه نیست. نمیتوانیم تضمین کنیم که به کم هزینه ترین هدف برسونند.

IDS: مشکل قبلی این بود که نمیتوانستیم خوب L را انتخاب کنیم. هعی یکی به یکی عمق را اضافه میکنیم تا به هدف برسیم. یعنی کلا چند تا DLS دارن اجرا میشوند. دقت کن هر بار L را افزایش میدهیم از اول الگوریتم اجرا میشود.

الگوریتم خوبی هست کامل هست قطعا بهینه هم هست. پیچیدگی حافظه آن خطی BD هست. چون تا عمق جواب دقیقا میرویم. ضریب انشعاب فقط باید کم باشد.

این نکته یادت باشه هر جا iterative باشد یعنی کار تکراری یعنی اضافه کاری.

الگوریتم جستجو دو طرفه: دو تا سرچ همزمان از حالت هدف یکی و از حالت استارت یکی و هر موقع بهم رسیدند یعنی تمام . کار تمام است و مسیر پیدا شده است. چرا خوب هست؟ چون عمق جواب کمتر هست هم از نظر حافظه و زمان مناسب هست دو طرفه این شکلی هست چون هر کدوم d/2 میرویم که خیلی بدرد ما میخورد دیگه یعنی d/2 خیلی کمتر هست دیگه. فقط باید بدانیم عکس هر اکشن را بدانیم یعنی باید اکشن ها برگشت پذیر باشند مثل گراف و یک مشکل دیگر این هست که خیلی از گراف ها اصلا هدف مشخصی ندارد مثل هشت وزیر. هدف را نداریم صرفا یک توصیف داریم و پس از حالت هدف نمیتوانیم اجرا کنیم و تست هدف هم مشکل داریم تست هدف این هست که گره مشترک در فرانتیر اینها وجود دارد یا نه و این چالش هاش هست.

فصل سوم بخش سوم:

در الگوریتم های اگاهانه یک سری اطلاعات خاص مسئله هم دریافت میکنند تا جواب بهتری تولید کنند. UCS به فاصله از مبدا نگاه میکرد فقط و حدودی نمیدانست تا هدف چه قدر فاصله دارد پس قرار هست یک تابع هیورستیک که تخمینی از یک گره تا هدف هست تا کمتر گره گسترش بدهند.

مقادیر تابع هیورستیک منفی نباشند و برای گره هدف باید صفر باشد مقدارش.

GBFS: اول حریصانه برابر با هیورستیک هست یعنی هر کدوم هیورستیک بهتری داشت همون. یعنی هر کدوم مقدار کمتری داشت H همون را انتخاب و آزمون هدف و گسترش میدهد. درختی هست این الگوریتم. مثلا توی اون بخارس هیورستیک میتواند خط صاف باشد بعد کوتاه ترین. خوب اگر درختی بود چون حلقه ایجاد میشود کامل نیست اگر درختی بود کامل هست. بهینه نیست.

A\*: هزینه مسیر از گره شروع به گره فعلی بعلاوه هزینه تخمینی هیورستیک تا گره هدف از گره فعلی مجموع این را به عنوان تابع سودمندی خود حساب میکند. و گره سودمندی تخمینی از ارزان ترین مسیر هست هر جا \* گذاشتیم یعنی واقعی یا بهینه. G یک گره میتواند تغییر کند حین حل مسئله ولی G\* یک مقدار ثابت هست. کامل هست اگر یک کران پایین مثبتی براش در نظر بگیریم. اگر قابل قبول بود یعنی هزینه تخمینی به ازای هر گره کمتر از هزینه واقعی بود اونوقت در جستجوی درختی بهینه هست. در برهان خلف دقت کن که قطعا گره ای در فرانتیر هست که روی مسیر بهینه بوده ولی بسط داده نشده هست و از قبل آن اشتباه رفته. F(G) همان مسیر ابی رنگ هست. مقدار هیورستیک برای گره هدف صفر هست پس در نظر نمیگیریم. چرا تبدیل به G\* شده؟ چون قطعا روی مسیر بهینه هست دیگه. پس میتوانیم جای G N بنویسیم.

اون سازگاری C هزینه بین اون دو گره هست. اگر سازگار باشد در حالت گرافی هم بهینه هست. سازگاری ویژگی قوی تری هست پس اگر سازگار باشد قابل قبول هست ولی برعکس نه. UCS همون A\* هست که هیورستیک سازگار دارد چون هیورستیک صفر هست و مقدار یال ها عددی بزرگتر از صفر هست. اگر هیورستیک قابل قبول نباشد نمیتوانیم بگوییم به مسیر بهینه نمیرسد.

کانتور ها در UCS دایره ای شکل هستند چون به مقدار G فقط کار دارد و برای اجرای A\* بیضی شکل میشوند و بیضی ها کشیده میشوند به گره هدف و میخواهیم بدونیم با برداشت کدوم گره ها به هدف نزدیک تر میشویم و هر چه قدر هیورستیک بهتر باشد بیضی ها سریعتر به هدف متمایل پیدا میکند. پس اگه نا اگاهانه یا دایره شکل بودیم همینطوری باید دایره رو زیاد میکردیم تا برسد به هر کدوم از این گره ها.

تا سر دقیقه 21 دیده شد.

الگوریتم a\* وقتی اجرا میشود گره هایی f آن بیشتر از هزینه مسیر بهینه باشد را گسترش نمیدهد و به نوعی هرس میکند.

A\*: وزن دار میگوید که ما تابع هیورستیک ما را در یک ضریبی ضرب کنیم تا سرعت ما بالاتر برود و تعداد گره های کمتر میشود و هزینه راه حل بدست آمده حداکثر w برابر هزینه راه حل بهینه هست. وقتی w بینهایت بشود عملا g n دیگه اهمیتی ندارد و ما مثل الگوریتم حریصانه میشویم. حتی اگر به ازای همه گره های هیورستیک قابل قبول نبود و فقط روی مسیر بهینه قابل قبول بود هم مسیر بهینه بدست میاد.

گفته بودیم اگر h از h\* بیشتر شود دیگر اهمیتی ندارد و مورد قبول نیست ولی اگر اختلاف h با h\* برابر باشد با تفاضل دومین مسیر بهینه منهای مسیر هدف تخمین بزند قطعا مسیر بهینه درست میاد. درختی قابل قبول باشد بهینه و گرافی سازگار باشد بهینه.

D: عمق جواب یا عمق مسیر بهینه.

فصل سوم بخش چهارم:

مقدار cut off با کمترین گره موجود اپدیت میشود. بدترین حالت کانتور این هست به ازای هر بیضی ما فقط یک گره را بررسی کنیم پس بزرگترین مشکل این الگوریتم این هست تعداد گره ای که تولید میکند خیلی زیاد هست.

RBFS: A\* هر چی تولید میکند را در حافظه نگه میدارد ولی در این الگوریتم گره هایی که بدرد ما نخورد از حافظه پاک میشود. در مسیر بازگشت مقدار گره با بهترین مسیر فرزند آن اپدیت میشود. گره هدف فراخوانی میشود و دیگر گسترش پیدا نمیکند همون لحظه تولید چک میکند. این الگوریتم در هر لحظه فقط یک مسیر را بررسی میکند و مسیر دوم آلترناتیو دارد که اگر بدتر بود بریم سراغ اون. این الگوریتم هم از تولید گره های تکراری رنج میبرد. بهینگی مثل a\* و حافظه آن خطی هست چون ما فقط یک مسیر را نگه میداریم.

SMA\*: دو الگوریتم قبلی حافظه را خطی نگه میدارند. این الگوریتم بر اساس اینکه چه قدر حافظه در اختیارش میدهیم بهترین جواب را با مثلا اون 10 تا گره پیدا کرد را میدهد و از اون حافظه دیگه الکی استفاده نمیکند. هنگام بازگشت گره با بهترین مقدار فرزند آن اپدیت میشود و ما مطمئن هستیم مسیر فعلی بهترین مسیر ممکن هست. دقت کن وقتی بسط یک گره تمام شد مقدارش باید با بهترین فرزندش اپدیت شود و همچنین وقتی حافظه پر شد باید بدترین انتخاب حذف و در پدرش مقدارش ذخیره شود.

اگر حافظه پر شد و یک برگ داشتیم و هدف نبود و هدف بهتری نبود باید از آن برگردیم و وقتی میخواهیم گره پدر را اپدیت کنیم مثبت بینهایت میگذاریم. اون بد بودن را با مثبت بی نهایت نشان میدهیم. وقتی هم برمیگردیم پدرش را بین بهترین فرزندانش مقدارش را اپدیت میکنیم. کلا برو ببین چه مسیر هایی که هدف دارند تو حافظه جا میشوند. بعد بین اونها بهترین مسیر. هیورستیک قابل قبول و سازگار باشد بهترین مسیر قابل ذخیره را برمیگرداند. همان مشکل تولید گره و این شاخه به اون شاخه پریدن را دارد. به طور کلی هیورستکی خوب هست که مقدار b\* را کمتر نگه دارد هر چه قدر هیورستیک بدتر باشد یا مثل BFS بریم ضریب انشعاب بالاتر میرود و دیرتر به جواب میرسیم.

استفاده از مسئله راحت شده: میایم مشابه مسئله اصلی یک مسئله راحت تر شده را بدست میاورم یک هیورستیک تا یک حد پایین برای هیورستیک واسه مسئله اصلی بدست بیاد و خیلی سریع باید این حد پایین بدست بیاد. یعنی مسئله راحت شده خیلی سریع باید حل شود. مثلا فاصله منهتن از آسمان نیومد اومدیم گفتیم دو تا کاشی رو میتوانیم با هم جا به جا کنیم بدون اینکه اون یکی کاشی خالی باشد یا نباشد که همین قضیه منهتن را به وجود آورد چون میتوانستیم یکی به یکی جا به جا کنیم.

کلا هدف این هست که ما h را به h\* نزدیک کنیم. اگر h1 و h2 قابل قبول باشند max انها هم قابل قبول هست.

حل زیر مسائل: زیر مسئله تعریف میکنیم. حل میکنیم همون هیورستیک میشود برای کل. مثلا یک زیر مسئله واسه هشت پازل این هست که واسه 4 تا کاشی این کار را بکنیم. بعد بیایم همه این مسائل را حل کنیم بعد واسه 8 تا ببینیم این حالت با کدوم یکی از حالت هایی که واسه 4 تا حل کردیم برابر هست بریم تو دیتا بیس و اون رو برداریم بیاریم. یک کار دیگه هم این هست که مثلا بیایم یکبار واسه 1 تا 4 و بار دیگر برای 5 تا 8 حساب کنیم بعد جمع بزنیم هیورستیک ها را آیا این قابل قبول هست؟ خیر. چون مثلا وقتی داریم واسه قسمت اول درست میکنیم یک مقداری از قسمت دوم هم درست شود پس بین مجموع این ها ممکن هست یک سری حالت مشترک باشد که 2 بار شمرده شده است پس لزوما قابل قبول نیست و کوچک تر از h\* نیست. اگر اون حالت ها دیگر رو در نظر نگیریم میشود که جمع قابل قبول باشد.

**فصل چهارم:**

مسائلی که در فصل قبلی بررسی کردیم در ساده ترین حالت ممکن بود یعنی محیط قطعی کاملا مشاهده پذیر و شناخته شده است. تو این بخش میایم همون ایده های جستجو رو روی محیط های غیر قطعی و نیمه مشاهده پذیر پیاده سازی میکنیم. الگوریتم های قبلی میومدند روی فضای حالت مسئله درخت جستجو را رسم میکردند و به صورت سیستماتیک جستجو میکردند و مسیر ها را در حافظه نگه میداشتند تا یک مسیری به هدف برسد و راه حل یک توالی از اکشن ها بود. در خیلی از مسائل جستجو ما دنبال توالی اکشن ها نیستیم بلکه حالت هدف داشته باشیم مثل مسئله 8 وزیر. اصلا مهم نبود اول کدوم 8 تا وزیر را میگذاریم. و اگر مسیر مهم نباشد از جستجو های محلی استفاده میکنیم این الگوریتم ها بین حالات مسئله جا به جا میشوند. محلی بخاطر همین هست چون به حالت های همسایه میروند و اصلی مسیری نگهداری نمیشود و سیستماتیک نیست و مزیت این هست که حافظه مصرفی خیلی کم هست. و هدف یافتن بهترین حالت برای اون هدف هست یعنی حالتی که در آن اون تابع بیشینه یا کمینه شده باشد مثلا از جنس هزینه باشد دنبال کمینه کردن هستیم.

دو بخش داریم: بخش جستجو های محلی. الگوریتم های حالت قبل الگوریتم های سیستماتیک بودند یعنی کل فضای حالت را از حالت شروع به شکل سیستماتیک شروع میکنند و دنبال یک مسیر تا رسیدن به هدف انتخاب میکنند. ولی جستجو های محلی دنبال مسیر نیستند و سیستماتیک نیستند و بین حالات جا به جا میشوند و بر خلاف فصل گذشته در بخش دوم درس دیگر خبری از محیط قطعی نیست و راه حل یک پلن مشخص نیست که از اول معلوم باشد و وقتی ما یک اکشن را انجام میدهیم نتیجه مشخص نیست چون محیط قطعی نیست و بحث اگر میاد وسط و پلن ما شرطی میشود.

جستجو محلی: ما دنبال این هست که ما به یک حالت هدف برسیم و مسیر برای ما مهم نیست و فقط میخوایم به اون حالت برسیم. و در حافظه در هر لحظه یک حالت را ذخیره میکنند و با حالت s مثلا همسایه ها مشخص میشود و سعی میکنیم به یک حالت دیگر برویم و بعد اون گره قبلی را پاک میکنیم و اینقدر تکرار میشود تا به هدف برسیم و مزیت آن این هست که از حافظه کمتری نگهداری میکنند. و برای محیط های بزرگ و نامحدود الگوریتم ها خوب عمل میکنند برخلاف الگوریتم های فصل قبل. مسئله بهینه سازی یعنی به حالتی برسیم که اون تابع هدف ما بیشینه یا کمینه باشد بسته به هدف ما. حالت مسئله ما یک بعدی هست فعلا، ماکسیمم محلی موقعیت هایی هستند که نسبت به همسایگی از خودشون اوضاع بهتری دارند و f بهتری دارند نسبت به همسایه ولی بهترین انتخاب نیستند. حالت flat هم یعنی همه همسایه ها یک f یکسانی دارند و از طرفی راهی به بالا نداریم. شانه: عین flat هستند ولی در ادامه میتوانیم به سمت بالا حرکت کنیم ولی جفت اینها f یکسانی دارند. دقت هم کن همین حالت عکس برای مینیموم هم میتواند اجرا شود.

جستجو تپه نوردی: در هر لحظه یک حالت را نگهداری میکنیم و همسایه ها را تولید میکنیم و به همسایه ای میرویم که مقدار f بیشتری دارد میرویم. و اگه هدف ماکسیمم باشد میشود صعودی ترین شیب اگر مینیموم هدف ما باشد میشود گرادیان نزولی. شرط این هست که ما باید بتوانیم همسایگی را تولید کنیم یعنی بتوانیم همسایه ها را تولید کنیم. همسایه ها همون حالت های بعدی هستند و یک تابع هدف داریم که به هر حالت یک مقداری نسبت میدهد و اونکه f بیشتری دارد را انتخاب میکنیم و همین روند را ادامه میدهیم. مثلا در هشت وزیر هر حالت مسئله یک چیدمان از هشت وزیر هستند بر روی اون سطح. و تابع هدف ما زوج وزیر هایی که همدیگر را تهدید میکنند. مقدار h را برای این حالت خیلی راحت بدست میاوریم. الگوریتم کاملی نیست ولی تا حد خوبی جلو میبرد. جواب معقول و سریع میدهد پس حریصانه عمل میکند. برای بهتر کردن میگیم که تو اگه به فلات رسیدی فعلا برو تا شاید این شانه باشد و به سمت همسایگی های بهتر بروی. و این مشکل میشود که تا ابد تو یک حلقه میفتیم. تپه نوردی تصادفی: قبلی میگفت اونکه 100 درصد بهترین هست را فقط برو ولی اینجا میگوید تو همه را در نظر بگیر به شکل احتمالی برو به یکی از اینها و اونکه اوضاع بهتری دارد را احتمال بیشتری بده و هر کی به اندازه سهم بیشتری که دارد شانس بگیرد. اولین انتخاب: همسایه را یک به یک تولید کن اولین حالت بهتری که دیدی نسبت به فعلی را انتخاب کن منتظر کلش نمان. تپه نوردی با شروع مجدد تصادفی: برخلاف قبلی ها میتوانیم بگوییم این الگوریتم کاملی هست و احتمال 1 کامل هست از یک جایی رندوم شروع میکند تپه نوردی اجرا میکند به هدف رسید که هیچ نرسید از یک نقطه دیگه شروع میکند به صورت تصادفی اینقدر این کار را میکند تا به حالت هدف برسد ولی ممکن است تعداد گام ها زیاد باشد. به احتمال 6/1 ام ممکن است عدد 5 بیاد یا همین قدر عدد 4 بیاد. و به طور متوسط اگر یک تقسیم بر یک دوم یعنی 2 بار این را اجرا کنید انتظار دارید که شیر بیاد. یعنی اگر احتمال 0.2 باشد یا 20 درصد ما 4 بار شکست میخوریم بار 5 ام موفق میشویم به طور متوسط.

فرموله سازی قبلی که مشکل داشت این بود که یک به یک وزیر اضافه کنیم. و تابع تعریف کنیم که خوب مشکل دارد چون ما داریم فرموله میکنیم ولی همه هشت تا وزیر روی یک سطح نیستند. و بخاطر همین نمیتوانستیم از جستجو محلی استفاده کنیم. جستجو محلی شرط این هست هر حالت مسئله تمام اجزای راه حل را دارد فقط مشکل این هست که اینها همدیگر را تهدید میکنند. بحث تعریف حالت های مسئله هم هست که برای هشت وزیر مثلا هر وزیر را در ستون خودش جا به جا کنیم. و اون موقع مثلا میتوانیم یک وزیر را انتخاب و در ستون خودش جا به جا کنیم پس هر حالت مسئله 56 همسایه خواهد داشت چون هشت تا وزیر هست و هفت تا در یک ستون میتوانند حرکت کنند. اون عدد ها هم به این شکل هست که مثلا 18 تا زوج وزیر به همدیگر تهدید میکنند و بهترین حالت 12 تهدید هست. کد هم ساده هست اگر بهتر از حالت فعلی نبود تمام میشود و اگر نبود گره فعلی را با بهترین جایگزین میکنیم پس خیلی حریصانه هست و تپه نوردی حریصانه نیز نام دارد. این سرعت یک سری مشکلات را دارد: اولین ماکسیمم محلی هست که ما تو اون حالت گیر میکنیم. و نمیتوانیم حرکت کنیم و تو بهینه محلی گیر میکنیم. بحث فلات ها هم هست و چون همسایه بهتر نداریم الگوریتم متوقف میشود فقط وقتی حرکت میکنیم که بهترین باشد. Ridge یعنی باید حرکت زیگزاگ بزنیم. یعنی اینکه حالت بهتری که نسبت به حالت فعلی هست ما مستقیم نمیتوانیم بهش برسیم و باید مثلا یکبار زیگزاگی پایین برویم بعد بالا که خوب در بهینگی محلی این حالت امکان پذیر نیست. در کل سریع به جواب میرسد. این مسئله مینیموم هست خیلی راحت به ماکسیموم تبدیل شود مثلا تعداد زوج وزیر هایی که همدیگر را تهدید نمیکنند. حالت ماکسیموم سازی ساده تر هست و انتخاب دو از هشت هست دیگه چون میخوایم زوج ها همدیگر را تهدید نکنند. یک مشکل خیلی عمده این هست که ما خیلی زیاد به فلات گیر میکنیم و الگوریتم متوقف میشود. در هشت وزیر. یک ویژگی این هست که ما sideways را اینقدر انجام بدهیم تا امیدوار باشیم در شانه باشیم و به سمت گره بهتر حرکت کنیم sideways یعنی اون گره با f مساوی هم انتخاب کن و حرکت کن به سمت آن. مشکل این هست که اگر در شانه نباشیم و در فلات باشیم کد هیچ موقع تمام نمیشود برای اتمام محدودیت میتوانیم بگیم مثلا 100 بار حق داری sideways بزنی و درصد موفقیت خیلی بالا میرود.

تپه نوردی تصادفی: برای حل مشکل بیش از حد بودن حریصانه حل شود. میگه به شکل احتمالاتی انتخاب کن صرفا به f بهتر انتخاب نکن. یعنی نسبت به تابع هدف به یک احتمالی برو اون گره به یک احتمالی به گره دیگر برو. احتمال h گره همسایه منهای h گره فعلی.

ویژگی این هست که دیرتر به هدف میرسد و احتمالاتی کار میکند و دیر تر همگرا میشود. یک ورژن دیگر تپه نوردی اولین انتخاب هست که به این شکل هست که حالت های همسایه را یکی به یکی تولید میکند و اگر یکی بهتر بود به همون میرود و باقی گره ها تولید نمیشود برخلاف استاندارد که اول تولید میکرد بعد بهترین رو انتخاب میکرد به درد مسائلی میخورد که همسایه ها خیلی زیاد هستند.

تپه نوردی با شروع مجدد تصادفی: صرفا میگه تپه نوردی عادی رو اجرا کن تا به هدف بهینه برسی و اگر نرسیدی از یک نقطه رندوم دیگه شروع کن تا به هدف برسی و همون هست فقط با حالت شروع رندوم و متفاوت. صحبت احتمالات میشود و 1/p بار باید این را اجرا کنیم تا به جواب بهینه اصلی برسیم بحث آماری هست. p احتمال موفقیت هست. همون توزیع هندسی هست و مقدار آن امید ریاضی هست. و مثلا یک تاس برای اینکه شش بیاد باید 1/6 بار بندازیم. مثلا تو هشت وزیر باید هفت بار اجرا کنیم. تا به هدف برسیم چون 14 درصد بود. بحث تعداد گام ها هم بود. تعداد گام ها اینطوری هست مثلا اگر هفت بار قرار هست اجرا شود و 6 بار قرار هست شکست بخوریم و در نهایت بار هفتم به جواب برسیم و برای شکست 3 گام و برای پیروزی 4 گام نیاز داریم میشود 6 ضربدر 3 بعلاوه 4. به طور متوسط هست.

Random walk: به طور حالت تصادفی برو به یکی از حالات همسایگی این خیلی غیر حریصانه هست یعنی در نقطه مقابل تپه نوردی هست و خیلی ناکار آمد هست فقط خوبیش کامل بودن هست برخلاف حریصانه شبیه سازی ذوب فلزات میاد وسط این 2 تا را میگیرد. شبیه سازی ذوب میاد اول تصادفی بودن را زیاد تر میکند اما هر چه قدر که میگذرد به انتها میرسیم به تپه نوردی متمایل میشود. تابع هدف یا e میشود سطح انرژی. T دما هست که اول زیاد بعد کم میشود. اول بد ها را هم انتخاب میکنیم اکتشاف زیادی میکنیم ولی به مرور حالات بد کم میشود. در هر دور به طور رندوم انتخاب میکند. به احتمال اون فرمول حالت بد انتخاب میکند اگر t بالاتر باشد اون عدد بزرگتر هست پس احتمال انتخاب هم بیشتر هست اگر شانس گرفت اجرا شد که هیچ اگر نشد رندوم یک همسایه تولید کن و همین مراحل را تکرار کن. کی تمام میشود؟ یا دما به 0 برسد یا به حالت هدف برسد دما چطوری کم شود دست خود طراح هست. دما رو خیلی نباید بیاریم پایین تا به احتمال 1 به ماکسیمم سراسری برسیم.

شبیه سازی ذوب فلزات: سعی میکند از اون ذات حریصانه تپه نوردی بکاهد. چون مشکل تپه نوردی این بود به بهینگی محلی میرسد دیگر متوقف میشود منتها در این شبیه سازی اجازه میدهد به حالت بدتر بروی البته با احتمالاتی یعنی به یک احتمالی اجازه میدهد که بروی به گره بدتر این احتمال هم هعی کمتر میشود. تا اواخر که شبیه تپه نوردی میشود. پس دو ایده تپه نوردی و قدم زدن تصادفی دارد. ما کلا دو طیف داریم که شدید ترین آن تپه نوردی هست که سریع هست ولی کارا نیست در مقابل قدم زدن تصادفی که حریصانه نیست و تصادفی حرکت میکنیم و کامل هست. با حرکت بد از بهینگی محلی خارج میشویم و وقتی دما یا t بیشتر احتمالا حرکت تصادفی بیشتر و کمتر شد احتمال حرکت تصادفی کمتر میشود. اگر دما صفر برسد باید حالت فعلی را به عنوان جواب برگردانیم. مثلا فرض کن ما تو گره ب هستیم میایم یکی از همسایه ها را تصادفی انتخاب میکنیم. حالا دو حالت دارد که گره پ از ب بهتر هست که در این صورت گره را انتخاب میکنیم ولی اگر پ از ب بدتر بود به یک احتمالی این را انتخاب میکنیم. به مقدار اون احتمال هم داخل جزوه هست. هر چه قدر دلتا e بدتر باشد ما حرکت بدتر داریم مقدار عدد کمتری میشود. و هر چه قدر t بیشتر شود احتمال انجام حرکات بد بیشتر هست. پس نگاه کن یکی از همسایه ها را به صورت تصادفی انتخاب میکنیم بعد میایم ببینیم اگر دلتا e عدد بیشتر از حالت فعلی بود انتخاب میکنیم اگر کمتر بود باید با احتمال کار کنیم.

اگر دما را به اندازه کافی کاهش بدهیم به احتمال ماکسیموم سراسری میرسیم.

جستجو پرتو محلی: همه این الگوریتم ها با یک حالت کار میکردند یعنی فقط یک حالت در حافظه داشتند و خیلی صرفه جویی میکردند ولی جستجو پرتو محلی k تا حالت تصادفی را درون حافظه در نظر میگیرد برای ساخت نسل k تا بعدی میاد همسایه تولید میکند مثلا 10 تا باشند هر کدوم 5 تا همسایه خوب میشود 50 تا گره میاد بین این 50 تا 10 تای بهترین را انتخاب میکند. و همین روند را ادامه میدهد تا به حالت هدف برسد اگر یکی از این همسایه ها هدف بود که هیچی. فرق این الگوریتم با تپه نوردی این هست که گره ها با هم همکاری میکنند و به هم اطلاعات میدهند. مثلا در تپه نوردی اول 16 بعد 120 انتخاب میشد ولی در پرتو محلی چون اون گره پ تولید شده شکل گرفته ما بهترین ها را برداریم 110 و 120 انتخاب میشود که بهتر هست. و گره ب را نادیده میگیرد و میرود سراغ اون اوضاع بهتر. البته همین قضیه مشکل هست چون تنوع را از دست میدهیم و به ناحیه کوچکی از فضای حالت برسیم. بهبود حالت همون این هست که تصادفی عمل کنیم و k تا برتر رو به شکل احتمالاتی انتخاب میکنیم و این احتمال با مقدار خوب بودن حالت همسایه هست. با تپه نوردی فرق میکند میاد بهترین همسایه ها را برمیدارد ممکن است مال یک نود باشد و اطلاعات مفید منتقل میکنیم به همدیگر و میگیم اینجا اوضاع بهتر هست همه بیاید اینجا. خیلی هم خوب نیست چون تنوع داشتیم و جاهای مختلف فضای حالت را سرچ میکردیم ولی یهو میایم یک تیکه خاص و انگار یک تپه نوردی داری اجرا میکنی. برای حل این مشکل حدودی میتوانیم بگوییم شروع تصادفی هم داشته باشد یعنی اوضاع بهتر را به احتمال بیشتر انتخاب کن.

الگوریتم های ژنتیک: یک نسخه شبیه به پرتو محلی هست یعنی با جمعیت کار میکند. بر خلاف الگوریتم های قبلی حالت ها با هم ترکیب میشوند و حالت جدید رخ میدهد. جهش همون حالت تصادفی هست که اون موارد رو یک تغییر کوچکی رو آن میدهیم. قسمت انتخاب بر اساس تابع fitness هست همون f خودمون فقط دقت کن بیشترین حالت که اینها همدیگر را تهدید نمیکنند مثلا 24 تا برای حالت دیگه 23 تا و ...، هر چه مقدار این تابع بیشتر احتمال انتخاب هم بیشتر. حالا مثلا تا یک حالتی میرسیم که میگیم این بهینگی محلی ما هست یا میتوانیم محدودیت بگذاریم بگیم تا 100 نسل ادامه بده. در ابتدا تنوع حالت ها بیشتر هست ولی هر چه قدر که میگذرد افراد مشابه تر شکل میگیرد چرا چون بر اساس شایستگی انتخاب میکنیم و اونها تکرار میشوند. مهم ترین فرق هم ترکیب هست. و اگر اون قسمت ها معنی دار باشند مثلا 3 تا اول همدیگر را تهدید نمیکنند میتوانند زودتر همگرا شوند. یا به جواب بهینه مایل شوند. مثلا معنی دار بودند میتوانیم همین زوج وزیر هایی باشند که همدیگر را تهدید نمیکنند دقت هم کن خیلی پارامتر دارد و این مشکل هست. اینجا میگیم 2 تا والد با هم ترکیب میشوند تا حالت جدید تولید شود والد همان حالت هست. مثل قبلی با مجموعه ای از k تا حالت تصادفی شروع میکنیم. و باید برای نشان دادن حالات مسئله باید رشته ای از یک الفبای محدود 0 و1 نشان دهیم. کل حالت مسئله مثلا میشود 8 تا 3 بیتی. هر رشته ما یک مقدار شایستگی یا fitness دارد. 3 مرحله دارد در مرحله 1 میگیم که هر کدام از این ژن ها چه قدر احتمال انتخاب دارند که این انتخاب بر اساس fitness هست یعنی میگیم اولین نفر را میخواهیم انتخاب کنیم کدوم را اونکه شایستگی بیشتری دارد. و دقت کن به احتمال چه قدر؟ کل fitness ها در مخرج و یک fitness همان رشته میشود درصد احتمال. این مقدار جفت وزیر هایی هست که همدیگر را تهدید نمیکنند مثلا 24 یعنی جفت وزیر هایی که همدیگر را تهدید نمیکنند. در مرحله ترکیب 2 تا والد را ترکیب میکنیم و این نوع دلخواه هست. مثلا یک نقطه را انتخاب میکنیم از اون به قبل از اولی و از اون به بعد از دومی. بعد رشته حاصل شده را ما هر کدام از ژن ها را به احتمال خیلی کمی تغییر میدهیم که مرحله جهش نام دارد این هم دست خود طراح هست. در مرحله ترکیب چون وقتی ترکیب میکنیم ممکن هست شبیه 2 تا والد خودش نباشد الگوریتم به اصطلاح گام های بلندی دارد و تنوع بالا هست. پس جمعیت اولیه متنوع هست ولی هر چه قدر که میگذرد چون تنوع کمتر میشود و اونایی که fitness بیشتری دارن باقی میمانند. و در نسل های بعدی تنوع کمتر میشود. و الگوریتم میاد بلاک های کاملا مختلف و متنوع را با هم ترکیب میکند دو تا تیکه مختلف و مستقل از هم و برای ساختن حالت هدف بهتر هست. و جستجو را در سطح ریز تری ادامه بدیم و حالت هدف را تیکه به تیکه کنیم و هر کدام را سعی کنیم تولید کنیم و بلاک های خوب ما در نسل های بعدی ادامه پیدا میکند. الگوریتم هم با یافتن هدف یا اتمام تعداد دور های مشخص تمام میشود.

دقت کن تابع دما باید نزولی باشد، و نباید مقدار دما زیاد شود هر چه قدر که میگذرد.

جستجو در محیط های پیچیده – فصل چهارم بخش دوم:

جستجو در محیط های غیر قطعی و نا مشاهده پذیر هست. نکته این هست که راحت از اول میتوانستیم حساب کنیم با یک سلسله اکشن به کدام مسیر و هدف میتوانستیم برویم و عامل میتوانست چشم هاش را ببندد مثل سیستم اپن لوپ و اون ادراک و پرسپشن ها تاثیری نداشتند.

در محیط نیمه مشاهده پذیر و غیر قطعی ادراک خیلی مهم هستند و عامل فقط قسمتی که هست را میتواند ببیند و بقیه را نمیتواند ببیند. این ادراک کمک میکند در هر لحظه تو چه حالتی هستیم و محدود میشود اگر غیر قطعی باشد نتیجه اکشن میتواند حالت های متنوعی داشته باشد و از ادراک میتوانیم استفاده کنیم و بفهمیم کدام حالت رخ داده است. و با شرط درگیر هست مثلا اگر فلان شد فلان بکن و یک برنامه ریزی جواب و استراتژی داریم و بر اساس ادراک دریافت شده اجرا میشود در واقع ما یک پلن داریم که اگر فلان ادراک را دریافت کردی فلان کار را بکن این میشود یک استراتژی. یک حالتی داریم که مثلا چشم بسته هست و هیچ سنسور نداریم. اما این مسائل خیلی راحت تر هستند و یک توالی از اکشن ها هست مثل قطعی و دیگه استراتژی نداریم. یعنی ما مطمئن هستیم قرار نیست چیزی به ما اضافه شود پس یک پلن میریزیم و میرویم جلو.

در نیمه مشاهده پذیر حالت ها محدود میشود و با استفاده از یک ادراکی از محیط میفهمیم در کدام حالت هستیم. غیر قطعی هم که همین هست با ادراک میفهمیم حاصل اکشن چی بوده هست. و هدف این هست یک عاملی طراحی کنیم که با استفاده از پلن که طراحی کردیم برود جلو مثلا اگر اینجا بودی این کار را بکن و طبق پلن اونکار را انجام میدهد.

در محیط غیر قطعی دیگه ما فقط به یک حالت نمیرویم به مجموعه ای از حالت ها میرویم. مثلا حالت غیر قطعی به این شکل هست که گاهی اوقات نه تنها تمیز نمیکند بلکه کثیفی قرار میدهد یا وقتی تمیز میکند کناری هم تمیز میشود این میشود غیر قطعی البته فرض کردیم مشاهده پذیر کامل هست. در قبلی ها عامل تعیین میکرد ما به چه حالتی برویم ولی اینجا محیط هم دخالت میکند که به چه حالتی برویم. چرا میگیم گره اند ؟ یعنی محیط هر حالتی که انتخاب میکند ما باید به جواب برسیم یعنی پلن باید ما را به جواب برساند و تضمین کند که محیط هر حالتی برسد به جواب میرسیم به or بر اساس اکشن های عامل هست مثل قبل مثلا انتخاب راست. همیشه بعد از یک گره or به اند میرسیم و پلن باید داشته باشیم. و باید ببینیم چه اکشنی قطعی و غیر قطعی هست. مثلا جارو کشیدن تنها حالتی غیر قطعی هست که کناری هم کثیف باشد. و یک توالی از اکشن ها دیگه نیست و درخت هست که برگ هاش باید هدف باشند و 3 تا ویژگی داشته باشند 1 هر or باید حاصل یک اکشن باشد 2 اند هر چی بود به جواب برسیم 3 برگ کلا باید هدف باشد.

فلش های قرمز راه حل هستند.

یک سری الگوریتم داریم که جستجو میکنند درخت ها را تا به هدف برسیم. الگوریتم اول DFS هست که عمقی جستجو میکند. و بازگشتی هست و روی ریشه که قطعا OR هست با یک تابعی به اسم OR سرچ را صدا میزند و داخل این تابع میاد به صورت بازگشتی تابع AND را صدا میزند و هر کدام از AND ها به صورت بازگشتی میاد اون پلن ما را بر میگرداند مثلا اگر محیط گفت راست تو پلن راست را برگردان و ... ، تا کی میرویم پایین؟ مثلا به یک حالتی رسیدیم که هدف هست و برگرد به بالا. یا اگر به یک حالت تکراری رسیدی شکست میخورد. و دیگر بررسی نمیکند چون به لوپ خوردیم. کلا به دور بخوریم شکست میخورد و همیشه تموم میشود چون بی نهایت نمیرسد چون یا بن بست هست یا هدف. بن بست گره تکراری نیست دقت کن و جستجو گرافی هست. یک راه حل این هست که برای مقابله با دور یک پلن بدهیم که دور هم داشته باشد مثل while مثلا اگر به یک گره برگشتی دوباره برو راست مثلا تا یکبار از شر این خارج بشی.

حالا به حالتی میریم که محیط نیمه مشاهده پذیر هست و به یک مجموعه حالتی میرویم که مثلا تو همون جارو برقی از اون یکی خبری نداریم و یک حالت باور داریم که چه مجموعه حالتی هستیم بر اساس ادراکی که دریافت کردیم. حالا چند دسته میشود مثلا مسائلی که هیچ ادراکی از محیط دریافت نمیشود و خوب این خیلی راحت هست ما میگیم تو یکی از این حالات هستیم و یک اکشن انجام میدهیم و تابع باور ما یک سری از حالت ها را خط میزند و یک سری از حالت ها میموند و ادامه میدیم تا به هدف برسیم. در واقع ما روی فضای حالت جستجو نمیکنیم روی حالت باور جستجو انجام میدهیم و باید مسائل را به حالت باور تبدیل کنیم تا محیط به تمام مشاهده پذیر تبدیل شود تا عامل نتیجه را بداند که کجا وارد میشود. البته دقت کن این سایز خیلی بزرگ میشود 2^n حالت برای حالت های باور از فضای مسئله میشود. اگر یک فضای حالت n گره داشته باشه. حالت اور در حالت باور شامل مجموع همه حالت ها هست. ممکن است بعضی اکشن ها تعریف نشده باشند ولی اگر ضرر منفی نداشته باشد میتوانیم در نظر بگیریم جز اکشن ولی اگر ضرر داشت فقط باید اشتراک اکشن ها را بگیری. نتایج اکشن ها هم اگر مشاهده پذیر باشد که متناظر جمع کل حالت های بعد اکشن اگر غیر قطعی باشد مجموع حالات که رخ میدهد جمع میشوند.

تست هدف به این شکل هست که یک حالت باور حالت هدف ماست که تمام حالت های گل مسئله در آن باشند. هزینه اکشن ها ثابت هست. حالت های دسترسی از حالت های باور هم مهم هست که به تعداد 2^n نیست.

مشاهده پذیر جزئی که ادراک از محیط دریافت میکند: در اینجا میتوانیم یک درکی از محیط بگیریم و حالت باور را اپدیت کنیم. و اینجا پلن هم داریم یک پلن احتمالاتی که اگر فلان درک را داشتیم چیکار کنیم. این قسمت هم مثل قبلی عمل میکنی برای تعیین حالت باور و ... فقط باید تابع نتیجه را کمی تغییر بدهی چون ادراک هم داری چون یک تابع اپدیت هم اضافه میشود. دقت کن بعد اپدیت برای تمام حالت ها چک میشود که بین ادراکی که داریم کدوم واقعا اتفاق افتاده است 4.5 اتفاق افتاده یا 9.5 یعنی ما اول 10 پول داشتیم یا 5. و همه اینها تابع ریزالت را میسازند. و تابع ریزالت چندین حالت دارد 4.5 بود فضای حالت فلان 9.5 بود فضای حالت فلان. بعد میتوانیم یک سرچ AND-OR بزنیم. چون جواب احتمالاتی هست. عامل همیشه باید حالت باور را همراه خود داشته باشد. فرق دیگر این هست که برنامه ها احتمالاتی هستند. دقت کن تا الان هر چی دیدم محیط آفلاین بود یعنی قبل اینکه به محیط پا بگذارند جواب داشتند. و عامل محاسبات میکرد. این سرچ ها یکم محاسبه میکنند بعد کار میکنند این 2 تا بحث را هم ترکیب میکنند. مخصوص محیط های ناشناخته هست و باید استفاده کنیم یعنی یک کاری بکنیم ری اکشنی بدست بیاوریم بعد محاسبه کنیم. و مثلا در اون مثال دقت کن عامل درکی از اکشن بالا رفتن ندارد و عامل باید انجام دهد تا بفهمد که این اکشن او را به این حالت میرساند و در خاطر نگه میدارد محیط مشاهده پذیر هست ولی چیزی راجب نتیجه اکشن ها نمیداند و هزینه این کار را هم نمیداند. هدف این است که با کمترین هزینه به هدف برسیم. مسائل اکتشاف یعنی دنبال هدف نیستیم فقط میخواهیم محیط را بشناسیم. دقت کن در DFS یا BFS چون جز محاسبات ما بود هزینه ای نداشت جابجایی برای ما ولی در الگوریتم های آنلاین به این شکل نخواهد بود.

تا 1:14 دقیقه از فصل چهارم بخش دوم دیده شد.

جستجو در محیط های ناشناخته هم داریم که هیچی نمیدانیم یعنی باید اکشن را انجام بدهیم تا بفهمیم نتیجه اکشن چی میشود به این مسائل اکتشافی هم گفته میشود. الگوریتم DFS خیلی شاخه به شاخه نمیشد چون گره ای که بسط میداد روی همان شاخه بود فقط مشکل این بود که چون هر مسیر را دو بار میبینی ضریب رقابتی 2 میدهد ولی بقیه بیشتر هست. جستجو محلی آنلاین: LRTA برای هر حالت یک تخمین از همین حالت تا هدف را تخمین بزنیم که مثلا همان تابع هیورستیک هست. از راست تخمین فعلی 3 واحد و از چپ 10 واحد هست پس میریم راست و مقدار حالت قبلی را اپدیت میکنیم چون اونجا فکر میکردیم تا هدف 2 تا هزینه میخواد ولی الان دیدیم نه 3 تا هزینه میخواد.

اما در جستجو های انلاین محاسبات و اکشن با هم قاطی میشوند و همزمان هستند و در محیط های پویا و نیمه پویا بهتر هست چون محیط پویا حین تصمیم گیری عامل محیط تغییر میکند پس رویکرد انلاین بهتر هست پس بهتر هست اول اکشن انجام بدهیم بعد محاسبات بکنیم. و حتی نتایج اکشن ها را نمیدانیم که بخواهیم از قبل محاسبات کنیم در محیط های غیر قطعی هم مناسب هست. یعنی به جایی اینکه نتایج اکشن و حالاتی که به وجود میاد را محاسبه کنیم که بفهمیم کدام حالت میرود همان را انجام بدهیم و متوجه بشویم واقعا به کدام حالت میرود جا محاسبات. مثل بچه ای که تازه به دنیا آمده هست. ما میدانیم چه اکشن هایی میتوانیم انجام بدهیم ولی نتایج آن را نمیداند. مگر اینکه واقعا اکشن را انجام بدهیم. فرض میکنیم محیط کاملا مشاهده پذیر هست. حرکت برگشت هم نمیدانیم. یک جدول نتیجه هم داریم که بعد از انجام هر اکشن هعی کامل تر میشود هزینه مسیر ها را هم داریم البته عامل نمیتواند استفاده کند تا وقتی که بداند حالت پ واقعا رخ داده است. مهم ترین نکته نمیدانیم خروجی تابع ریزالت چی هست تا موقعی که انجام بدهیم. هزینه مسیر آنلاین کل هزینه ای هست که عامل آنلاین کلا طی میکند یک best cost هم داریم که هزینه کوتاه ترین مسیر تا هدف هست که کمتر از هزینه انلاین هست. در محیط های بن بست داریم ممکن است این هزینه ها بی نهایت شوند. کلا اکشن های غیرقابل برگشت داشته باشیم امکان بن بست و بینهایت شدن هزینه آنلاین وجود دارد. هیچ الگوریتمی نمیتواند از بن بست جلوگیری کند. برای اینکار میتوانیم به این شکل یک ناحیه امن در نظر بگیریم یعنی هدف ما قابل دسترسی هست. در افلاین ما یک شبیه سازی انجام میدهیم ولی در سرچ آنلاین حتما باید داخل یک حالت باشیم تا بتوانیم اون اکشن را انجام بدهیم. پس خوب هست یک الگوریتمی انتخاب کنیم که به شکل لوکال باشد تا هزینه زیاد نشود. پس پیاده سازی الگوریتم DFS به صورت آنلاین خیلی بهتر هست دیگه چون هزینه کمتری دارد و حتما باید بدانیم که اکشن ها قابل بازگشت هستند. یک جدول به نام ریزالت هم داریم. که نتایج اکشن را در این میریزد. چرا بهینه هست؟ چون هر یال دقیقا دو بار در بدترین حالت پیمایش میشود و از مرتبه عدد ثابت هست.

لوکال سرچ هایی که به صورت آنلاین میتوانند پیاده سازی شوند: تپه نوردی خودش یک الگوریتم آنلاین هست یعنی از یک حالت همسایه ها را در نظر میگیریم و به یکی از حالت ها میرویم فقط شروع تصادفی نمیتوانیم داشته باشیم و حتما باید به اون حالت برویم. حالت شروع تصادفی میتواند کند باشد بلاخره به هدف میرسد ولی. چون تعداد نمایی حرکت نیاز هست تا به هدف برسیم. و برای همین LRTA\* میرسیم. این روش بهترین مقدار که همان هیورستیک هست را ذخیره میکند. و وقتی رفتین به حالت بهتر هیورستیک فعلی را با همون هیورستیک کم اپدیت میکنیم. چون ما تو مینیموم محلی هستیم در تپه نوردی گیر میکنیم ولی در این الگوریتم گیر نمیکنیم.

**فصل پنجم بخش اول:**

جستجو خصمانه: همه مسائلی که برای جستجو بررسی کردیم همه محیط ها تک عامله بودند و یک سری حالت مطلوب به نام هدف داشتند و دنبال یک پلن برای رسیدن به هدف بود ولی در این فصل با محیط های چند عامله برخورد میکنیم که رقابت میکنند برای سود بیشتر. و سود عامل ها با ضرر یکی دیگر باشد بازی نام دارد. معمولا نوبتی هستند و در یک لحظه چند تا عامل همزمان کار نمیکنند. این بازی ها zero-sum هست یعنی وقتی یکی میبرد یکی میبازد و مجموع امتیاز ها ثابت یا 1 واحد هست. و برد، برد نداریم و با اطلاعات کامل و کامل مشاهده پذیر هست یعنی رقیب میداند اون یکی چه تصمیم گرفته و چه تصمیمی میتواند بگیرد. MAX بازی را شروع میکند. تابع TO-MOVE میگوید الان نوبت کدام بازی هست. IS-TERMINAL یعنی حالتی که به عنوان ورودی میگیرد حالت نهایی هست یا نه. Utility همان تابع سودمندی هست. باید فرض کنیم حریف همیشه بهترین حرکت را انجام میدهد. مقدار mini max چیست؟ همون چیزی که کنار گره ها نوشتیم و مقدار سودمندی تابع max در آن بیشینه هست به شرطی که حریف min بهینه بازی کند به همین دلیل هست که مقدار اون شده 3 در واقع ماکس گرفتیم بین اون 3 تا. این واسه برگ ها هست اگر برگ نبود ما میایم بین گره های فرزند اگر نوبت max بود max میگیریم و اگر نوبت min بود و طبیعتا بین گره های max ما min میگیریم. اگر بدانیم حریف ما بهینه نیست اونوقت استراتژی minimax بهترین استراتژی ممکن نیست زیرا میتوانیم تله بگذاریم و بیشتر سود کنیم. دقت کن اون درخت بازی رو ما از اول نداریم و الگوریتم مینی مکس هست که به صورت عمقی مانند دفس درخت را جستجو میکند و این هزینه ها و درخت را برای ما میسازد. و هزینه مینی مکس را بدست میاورد. مقدار گره مکس در شروع منفی بینهایت و گره مین در شروع مثبت بینهایت هست. منفی بی نهایت ها قرار هست زیاد شوند و مثبت بی نهایت ها قرار هست کم شوند در واقع مثبت بی نهایت باید کمترین مقدار موجود باشد. یعنی اولین برگ هزینه سمت چپ را میگذاریم بعد با بقیه مقایسه میکنیم. تا کمترین را برداریم. بعد از تعیین و تکلیف اول گره فرزند یا همون برگ از سمت چپ مقدار ریشه یا گره پدر را با همان مقدار اپدیت میکنیم حالا باید دنبال این باشیم که بیشترین مقدار را بگیریم. تو بازی ها بیش از 2 دو بازیکن هدف این هست هر کسی به بیشترین سودمندی خود برسد و کاری به سود و ضرر بقیه ندارد. و در واقع نتیجه مینی مکس برای گره A میشود یال a1 زیرا بیشترین سود ممکن را برای این گره دارد. البته یک سری شرایط دیگر در بازی های بیش از 2 بازیکن وجود دارد مثلا اینکه ممکن است 1 بازیکن دیگر شرایط خیلی بهتری از 2 تای دیگه داشته باشد و 2 تای دیگه با هم همدست شوند و به نوعی انتخاب کنند که نفر اول که شرایط بهتری دارد ضرر کند. و دیگه استراتژی بهینه minimax نیست چون فقط به خودمون نگاه میکنیم و باید استراتژی های دیگری را انتخاب کنیم. دقت کن وقتی مینویسیم max مثبت 1 یعنی min منفی 1 هست ولی نمینویسیم.

هرس درخت های بازی: گفتیم مشکل اصلی الگوریتم مینی مکس این هست که b^m گره تولید میکند و خیلی زیاد هست. الفا و بتا از گره پدر به ارث میرسد. اگر هرس نشد مقدار الفا و بتا اپدیت میشود الفا میخواهد زیاد و بتا میخواهد کم شود. دقت کن اگر محدودیتی نداشتیم منفی بینهایت و مثبت بی نهایت هست واسه چی اول همه گره ها منفی و مثبت بی نهایت هست ؟ از پدر به ارث میبرند. واسه انتخاب min باید ببینیم که عدد کوچک تر از الفا هست یا نیست اگر بود هرس رخ میدهد اگر نبود مقدار بتا اپدیت میشود و واسه max باید ببینیم عدد انتخابی بزرگتر از بتا هست یا نیست اگر بود هرس رخ میدهد.

**فصل پنج بخش دوم:**

تعداد هرس به شروع پیمایش درخت بستگی دارد بهترین حالت این هست که ما حالت ها با مقادیر کوچک تر را زودتر بررسی کنیم. دقت کن اگر درخت 3 سطح دارد سه منهای یک سطح باید بررسی شوند ولی سطح آخر میتواند هرس شود و میشود از مرتبه B^M/2. بر خلاف الگوریتم مینی مکس که به گره های مکس میگفت بیشترین فرزندان و به مینی میگفت کمترین فرزندان و برگ ها هر مقداری دارند همان utility باشد در این الگوریتم شبیه به IDS میگوییم بیا به هر گره یک عدد نسبت میدهد حتی اگر حالت ها حالت نهایی نباشند. یک تابع ارزیابی میدیم و مقدار مینی مکس را حدس میزند و تقریبی هست. به جای DLS از IDS استفاده میکنیم تا عمق را یکی به یکی بریم پایین و بهترین جواب ممکن را گزارش کنیم. مثلا بیایم بگیم ما دو تا فیل داریم ولی حریف 1 فیل دارد بعد خود این فیل یک وزنی دارد دیگه این مثلا یک تابع ارزیابی تقریبی برای شطرنج هست. یعنی وزن هر مهره در تفاوت تعداد مهره هایی که من با حریفم دارم. و هر چه قدر تابع ارزیابی بیشتر ما نزدیک تر هستیم به بردن. تابع ارزیابی باید مقادیری که به برگ ها نسبت میدهد باید به ترتیب سودمندی باشد کلا ترتیب مقادیر برگ ها در مینی مکس هم مهم هست. ضرب یک عدد ثابت در مقدار گره ها ترتیب آنها را بهم نمیزند. پس باید مقادیری که نسبت میدهیم ترتیب را حفظ کند. یعنی به همون ترتیب سودمندی واقعی آنها باشد چون داریم تخمین میزنیم دیگه. مشکل اصلی این قطع کردن درخت جستجو عدم تضمین بهینگی هست. مشکل دیگر این هست که اگر جستجو رو روی حالت هایی که غیر ساکن هستند تموم کنیم اون موقع الگوریتم ممکن هست خیلی اشتباه بکند. حالت غیر ساکن حالتی هست که ممکن هست اون اطراف تغییر زیادی داشته باشد یعنی پدر خوب باشد مقدارش اما فرزندان خوب نباشد و ما متوجه نمیشویم و فرض میکنیم حالت خوبی دارد. اثر افق: حریف یک حالت خیلی خوبی دارد و اجتناب ناپذیر هست ولی بتوانیم اینقدر تاخیر بندازیم که اون حالت خوب بره تو گره های پایین تر که ما نمیبینیم.

بازی های تصادفی: تا الان همه چیز قطعی بود.

گره شانسی با دایره نمایش داده میشود و خروجی ها به صورت یال از آن خارج میشوند. اگر در یک گره احتمالی نوشته نشود یعنی احتمال همه یکسان هست. و باید میانگین وزن دار فرزندان بگیریم تا بتوانیم مقدار مینی مکس را بدست بیاوریم یعنی مقدار ضربدر احتمال. و بعد که امید ریاضی بدست اومد بین اونها اگه گره مکس بود مکس اگر مین بود مین.

دقت کن در بازی های قطعی تابع ارزیابی خوب هست که ترتیب کم به زیاد برگ ها را حفظ کند لزومی ندارد دقیق حدس بزند همچین چیزی برای عنصر شانس هم داریم. اما در عنصر شانس صرفا حفظ ترتیب برای ما کافی نیست و نتیجه خروجی متفاوت باشد با نتیجه اصلی. چون دقیقا مقدار شانس برای ما مهم هست و تبدیل خطی مثبت از مقادیر سودمندی باشد. یعنی نمایی نباشد.

N تعداد نتایج برای گره های شانسی هست مثلا پرتاب سکه 2 هست تاس 6. و دقت کن در شانسی ها ما هرس الفا بتا نداریم چون عنصر شانس دخیل هست و فقط برای قطعی ها داریم که میتوانیم مقدار مینی مکس را محاسبه کنیم. مگر اینکه مقادیر مینی مکس را به ما بگوید اونوقت میتوانیم احتمال بگیریم که مقادیر تو چه بازه ای هست. و هرس انجام بدهیم البته ایده هرس را میتوانیم داشته باشیم. یعنی هر موقع مطمئن شدیم مقادیر یک گره نمیتواند مقادیر بالایی را عوض کند هرس میکنیم. دقت کن اگر مقدار روی گره ها نباشد پس احتمال مساوی هست. ببین ایده اینطوری هست تو مکس نگاه کن سقف چه قدر سود تو مین نگاه کن کف چه قدر سود. چون میخواهیم ببینیم اینقدر خوب یا بد هست که ریشه را عوض کند یا نه.

**فصل ششم بخش اول:**

ارضای محدودیت طیف وسیعی از مسائل کامپیوتر را شامل میشوند و با تکنیک های جستجو و چیز های جدید در زمان بهتر و کارا تری مسائل را حل میکنیم. یعنی به مسئله ارضای محدودیت یک سری متغیر داریم با یک سری مقادیر و دنبال این هستیم که اون محدودیت ها ارضا شوند تحت مقدار دهی و شرایطی. دقت کن اینجا هم به ازای مقداری که میدهیم یک حالت جدید ایجاد میکنیم.

مسائل تا امروز اتمیک بودند یعنی مثلا در فلان شهر بودیم و چک میکردیم که این حالت هدف هست یا نیست و در مسئله چیزی از خود حالت ها نمیدانستیم و داخل حالت ها نمیدانستیم چی هستند به این میگویند اتمیک ولی در ارضای محدودیت به این شکل نیست و به ما هر حالت یک سری مقادیر دادیم به متغیر ها. یعنی مثلا اینکه هر متغیر چه مقادیری گرفته کمک میکند که حالت بعدی بهتری تولید کنیم. هیورستیک های قبلی خاص همون مسئله بودند و برای باقی مسائل فایده ای نداشت ولی در ارضای محدودیت اگر درست فرموله کنیم اون هیورستیک برای همه مسائل مورد استفاده هست مثلا همیشه متغیری را انتخاب کن که محدوده کمتری دارد و به این میگویند هیورستیک های عمومی.

هر متغیر یک دامنه دارد و مشخص میکند اون متغیر چه مقادیری میتواند بگیرد و یک سری محدودیت هم داریم و یک جواب از مسئله ارضای محدودیت این شروط را ارضا میکنند. حالا مثلا این شرط میتواند به صورت تابع باشد مثلا الف با ب برابر نباشد. یک راه دیگر این هست که همه ردیف های مجاز مقادیر را نشان بدهیم که معمولا از همون روش اول استفاده میکنیم. هر حالت مسئله یک مقدار دهی به یک سری یا همه متغیر ها هست. البته به یک سری از متغیر ها درست تر هست چون این خوب تر هست چون یک به یک مقدار دهی بکنیم لزومی ندارد به همه متغیر ها مقدار بدهیم. مثلا تو فصل سوم به جای اینکه اون 8 تا رو فرموله بندی کامل کنیم یعنی هشت تا وزیر که روی صفحه هستند ما چینش متفاوت بدیم برای هر حالت. فرموله سازی افزایشی به این شکل هست که ما وزیر ها را یکی به یکی اضافه کنیم که این بهتر بود در اینجا هم باید فرموله بندی افزایشی انجام بدهیم و نیاز نیست همون اول به همه متغیر ها مقدار دهی بکنیم. مثلا تست هدف این هست که مقادیر به نوعی باشند که هیچ محدودیتی نقض نشود. این مسائل تو زمان کم قابل حل نیستند ولی با جستجو به این شکل کارا تر هستند. دقت کن وقتی میخواهی به صورت explicit شرط را بنویسی باید برای همه حالت ها بنویسی. CSP باینری یعنی اینکه هر یال حداکثر دو متغیر را به هم وصل میکند چون محدودیت رنگ حداکثر بین دو تا متغیر هست که نباید رنگ یکسان داشته باشند ولی مسئله اون مقدار دهی اعداد دیگر گراف ساده نیست و بین 3 تا متغیر هست و دیگر گراف ساده نیست. در معماری اعداد هر حرف باید مقدار متفاوت داشته باشد. اگر دامنه مقدار دهی به متغیر ها محدود باشد از مرتب O d^n هست هر متغیر d تا حالت دارد و n تا متغیر داریم و جز مسائل np complete هستند و راه حلی با زمان n نداریم یا چند جمله و نمایی هستند اگر دامنه مسئله گسسته و نامحدود باشد اولا باید حالت ها را به صورت ضمنی نمایش بدهیم و مسائل زمان بندی کار ها را داریم چون مثلا کار ها باید به ترتیب اجرا بشوند و این مسائل اگر محدودیت به شکل خطی باشد این مسائل قابل حل هست البته خیلی بیشتر از نمایی هست ولی اگر محدودیت ها به شکل نمایی باشند مسائل بسیار سختی هستند و قابل حل نیستند. اگر متغیر های پیوسته باشد اگر این محدودیت به شکل خطی باشد میتوانیم در زمان چند جمله ای و به صورت کارا حل کنیم با استفاده از الگوریتم های lp. طیف وسیعی از مسائل را میتوان به عنوان CSP در نظر بگیر چون ما یک سری کار داریم و باید به ترتیب اجرا بشوند و تا حالت هدف که شروط ارضا شوند ادامه میدهیم و مسائل تخصیص و انتساب دادن و یا مسائل طراحی مدار باید از CSP استفاده میکنیم. حالا چطور میتوانیم مسائل CSP را با جستجو حل کنیم: ساده ترین نحوه فرموله سازی همون فرموله سازی افزایش هست که تو هر سطح از درخت میایم یکی به یکی به متغیر ها مقدار دهی میکنیم. جواب هم قطعا تو عمق N هست چون تو هر سطح داریم به یک متغیر مقدار میدهیم و اگر عمق را بدانیم میتوانیم از DFS یا عمق اول استفاده میکنیم تا درخت جستجو CSP را بسازیم. چون دور هم وجود ندارد با استفاده از یک به یک متغیر دادن. فرمول تعداد برگ N!.D^N هست که عدد بسیار بزرگی هست. و برای جستجو درخت خیلی طولانی هست و این نحوه فرموله بندی ایراد بزرگ آن زمان زیاد میشود و تعداد برگ ها خیلی هست. کار دیگر این هست که در هر سطح فقط به یک متغیر مقدار دهی بکن و به گره های دیگر کاری نداشته باش پس D تا حالت داریم دیگه. این نحوه بهتر از حالت قبلی هست چون ترتیبی برای متغیر ها قائل نبودیم و کلی حالت الکی داشتیم. و جایگشت از بین میرود و تعداد برگ ها هم فقط همان D^n میشود. فقط باید دقت کنیم در سطوح جدید محدودیت ها را رعایت کنیم و نقض نکنیم. مثلا اگر یک ایالت قرمز بود دیگر تو سطح بعدی فرزندان آن رنگ قرمز ندارند. این بحث نمایش عامل که چه مقداری گرفته است باعث شد جستجو سریع تری انجام بدهیم چون دیگر قرمز را تولید نمیکنیم و هرس میکنیم و با فرموله کردن مسائل به شکل CSP میتوان بخش زیادی از درخت را هرس کرد چون وقتی در یک شاخه که هستیم مثلا قرمز داریم که یک انتساب جزئی هست میدانیم که قرمز نباید باشد پس کل اون شاخه هرس میشود. و ترکیب DFS با این 2 تا ایده (1. در هر سطح فقط به یک متغیر مقدار بدهیم 2. هر انتساب انجام میشود صرفا در فرزندانی تولید کنیم که با انتساب فعلی محدودیتی را نقض نکند.) میشود الگوریتم عقب گرد یا backtracking. حالا برای کارایی جستجوی عقبگرد یا افزایش سرعت باید این کار ها را انجام بدهیم: مورد اول این هست که به کدام متغیر باید زودتر مقدار دهیم و مورد بعدی این هست که مقادیر یک متغیر باید به چه ترتیبی بررسی شوند یعنی مثلا شهر رنگ قرمز بگیرد زودتر بررسی شود یا رنگ سبز بگیرد مورد بعدی همون هرس کردن هست یعنی خیلی زودتر از عقبگرد بفهمیم این شاخه تهش هدف نیست که این بحث فیلتر کردن هست تا دامنه ها رو فیلتر کنیم و نکته بعدی این هست که آیا میتوانیم از ساختار مسئله که محدودیت دارد بررسی کنیم تا زودتر به جواب برسیم حالا سوال اول این هست که ما همون ترتیب دهی به متغیر هایی که باید مقدار بگیرند چگونه هست اینجا یک سری هیورستیک های عمومی معرفی میکنیم اولی MRV هست، میگه اون متغیری را انتخاب کن که مقادیر مجاز کمتری دارد. مثلا در اولین قدم این هیورستیک تاثیری ندارد چون همه متغیر ها یک دامنه ثابت دارند ولی در گام دوم چون مثلا رنگ قرمز دادیم یکی از دامنه کم میشود عقبگرد که اصلا رنگ تکراری را حذف میکند. یک حالت دیگر این هست که همون اول میرود سراغ اون گره ای که محدودیت بیشتری دارد و باعث میشود اون شاخه هایی که حذف میشوند خیلی بهتر هست چون میفهمیم به جواب نمیرسیم و هرس میکنیم. پس اول باید بریم سراغ گره ای که بیشترین محدودیت را دارد که سیاست first fail هست اما در شرایطی که MRV تعیین کننده نیست از هیورستیک درجه استفاده میکنیم که میگه برو سراغ اون متغیری که تعداد همسایه مقدار نگرفته بیشتری دارد. و باعث میشود ضریب انشعاب کمتر شود و همون اول میریم سراغ اونها که بیشترین فرزند را دارند.

دقت کن که اول درخت حل مسئله CSP را با فضای حالتی که داریم رسم میکنیم بعد جستجو میکنیم به کمک الگوریتم های جستجو. الگوریتم backtracking: یک جستجو عمقی و کامل هست با استفاده از این فرموله بندی و جواب قطعا در عمق n هست و گره تکراری نداریم ترتیب متغیر ها را فیکس میکند و شاخه هایی میسازد که سازگار هستند با مقدار فعلی. هیورستیک یعنی ترتیب مقدار دهی متغیر ها را مشخص میکند و همچنین عام منظوره هستند. دقت کن اگر بگوید همه راه حل ها دیگر اینکه به کی اول مقدار بدهیم فرقی ندارد ولی اگر بگوید فقط یک کار میتوانیم از LCV استفاده کنیم.

حالا میرویم سراغ این بحث که به چه ترتیبی دامنه یک متغیر را بررسی کنیم: حالا یک متغیر انتخاب شد به چه ترتیبی باید دامنه را انتخاب کنیم که باید از هیورستیک LCV استفاده کنیم که میگوید اولین مقداری را انتخاب کن که کمترین محدودیت را برای سایر ایجاد کند. در حل مسائل CSP یا ما دنبال همه جواب مسئله ها هستیم یا صرفا دنبال یک حالت اگر دنبال همه حالت ها باشیم چون قرار است همه درخت را جستجو کنیم دیگر اهمیتی ترتیبی ندارد. اما اگر صرفا دنبال یک جواب باشیم اول میرویم سراغ شاخه ای که کمترین محدودیت ایجاد کند. و LCV به ما کمک میکند.

**فصل ششم بخش دوم:**

هدف فیلتر کردن هرس کردن درخت هست. در مرحله بعدی میخواهیم نسبت به ساختار های خاص الگوریتمی بسازیم که الگوریتم سریعتر حل شود. انواع محدودیت ها در مسائل داریم. یکتایی : فقط شامل یک متغیر هستند. دو تایی یا باینری یعنی جفت متغیر مثلا دو شهر نباید رنگ یکسان داشته باشند. مرتبه بالاتر: شامل 3 تا یا تعداد بیشتر هست مثلا عدد بین 3 تا عدد میتواند باشد. محدودیت های عمومی شامل تعدادی دلخواه متغیر هستند و مثل alldif که میگوید هر چی هست باید متفاوت باشد دیگه کاری ندارد چند تا هست. یک سری محدودیت های ترجیحی یا نرم داریم که محدودیت های قطعی نیستند ولی ترجیح میدهیم رعایت کنیم. یعنی مثلا یک ایالت قرمز شده است میتواند آبی هم باشد ولی قرمز را ترجیح میدهیم به ابی مثلا یک استاد دانشگاه ترجیح میدهد صبح کلاس نداشته باشد حالا اگر داشت کلا غلط نیست ولی ترجیح این هست که اینگونه نباشد. و اگر این را مدل کنیم به گونه ایکه این محدودیت ها یک هزینه داشته باشند اونوقت دنبال کم هزینه ترین مسیر هستیم پس میشود یک مسئله بهینه سازی. همه محدودیت های با متغیر های زیاد تر را میتوانیم به متغیر های دوتایی تبدیل کنیم. خوبی این هست که ما تمرکز میکنیم فقط روی حل الگوریتم های دودویی و دیگر به انواع محدودیت ها فکر نمیکنیم. بدی این هست که محدودیت نوشتن سخت هست توی عمومی ها کلا یک alldif مینویسیم. ولی دو به دو سخت تر میشود. حالا یک سری الگوریتم داریم که برای محدودیت های عمومی بهتر هستند. فیلتر کردن: بیایم اون دامنه ها متغیر هایی که هنوز مقدار نگرفته اند در هر لحظه اپدیت کنیم و اونهایی که نمیتوانند باشند را حذف کنیم. هر چی دامنه کوچکتر سرعت بیشتر. و اگر اندازه یک دامنه صفر شود میفهمیم در اون شاخه جوابی وجود ندارد و حذف کنیم. این forward checking نام دارد که پس از هر انتساب میاد مقادیر ناسازگار را از دامنه ی متغیر های همسایه حذف میکند. مشکل این الگوریتم این هست که به شدت محلی عمل میکند چون فقط سراغ همسایه ها میرود و اونها را بررسی میکند و این محدودیت را منتشر نمیکند و این قدرت کافی را برای تشخیص محدودیت ندارد. و اگر منتشر میکرد بهتر بود چون همون لحظه عقبگرد انجام میدهد. دقت کن قدرت forward checking محدود هست چرا چون محدودیت های دو به دو را فقط چک میکند به سایر یال ها کاری ندارد.

مفهوم سازگاری یک یال: یک یال در گراف محدودیت سازگار هست اگر و فقط اگر به ازای هر مقدار x در دامنه X یک مقداری هم مثل y در دامنه ی Y وجود داشته باشد. که با آن سازگار باشد. یعنی محدودیت را نقض نکند. یعنی اینور بود چک کن اونور هم سازگار هست یا نیست. اگر وجود داشته باشد که هیچی اگر وجود نداشته باشد اون یال اون مقدار را از اولی حذف میکنیم. این کار همزمان با جستجو امکان دارد. زودتر میفهمیم با این روش مسئله جواب ندارد. اینکار forward checking نمیتوانست انجام دهد چون دو به دو بود. یک نکته مهم اگر یک مقداری را از دامنه حذف کردی دوباره باید گره هایی که یال مشترک با گره ای که دامنش کم شده دارند چک کنی که سازگار هستند یا نه. نتیجش این میشود که اگر مثلا به Q سبز دادی دیدی که یک گره دامنه آن خالی شد پس ناسازگاری تشخیص داده شد. و هرس شد درست هست هزینه نسبت به فوروارد چکینگ ( که 2 به 2 کلا میگرفت) بیشتر هست ولی این باعث میشود بحث هرس کردن درخت سریع تر شود تو بحث جستجو قبل از شروع هم میتوان اعمال شود با این روش. دقت کن سازگار کردن باید دو طرفه باشد حتما. دقت کن اگر از یک دامنه دوباره حذف شد باید تمام یال ها را دوباره چک کنی. دقت کن به ازای تمام مقادیر دم باید یه متغیر سازگار باشد اگر نبود درجا اون رو از دامنه دم حذف کن.

الگوریتم AC3: برای سازگار کردن گراف هست میاد یک گراف میگیرد به عنوان ورودی دونه به دونه یال ها برمیدارد میریزد داخل صف و یک به یک چک میکند سازگاری را و سازگار میکند نسبت به اون یکی سرش. دقت کن فقط به دامنه مبدا دست میزنیم. اگر دامنه تغییری نداشت ادامه میدهیم ولی اگر مقداری حذف شد دوباره باید همه یال هایی که با این گره ارتباط داشتند به صف اضافه کنیم. دقت کن گره هایی که دامنه تغییر میکنه و یال های متصل به آن دوباره در ته صف میروند تا سازگاری آنها چک شود. دقت کن ممکن هست یک مسئله ای جواب نداشته باشد ولی AC3 را اجرا کنیم دامنه ای حذف نشود. دقت کن به ازای تمام دامنه های head ممکن است یال آن با دم به صف اضافه شود حداکثر چه قدر ؟ d بار که اندازه دامنه هستش.

دقت کن جوابی حذف نمیشود صرفا جواب هست با دامنه های کوچکتر، الگوریتم AC3 قبل از جستجو رخ میدهد و کمک میکند جستجو سریع تر شود. ترتیب بررسی گره ها اهمیتی ندارد و در هر صورت خروجی یکسان هست. اونایی که از قبل بودند رو دیگر لازم نیست اضافه کنی که دوباره بررسی شود. کنکوری تر به این شکل هست که برو سراغ اونها که دامنه کوچکتری دارند. حداکثر به اندازه دامنه یک یال را اضافه میکنی. چک کردن سازگاری هر یال d^2 هست چون اندازه d تا تعداد داریم و در بدترین حالت به تعداد d به صف اضافه میشوند. الگوریتم MAC: پس از انتساب مقدار به متغیر الگوریتم AC3 را با یک صف حاوی یال های مقدار نگرفته هست اجرا کن. همان قبلی هست فقط به جای همه یال ها بیا اون یال هایی بریز که همسایه یال های مقدار گرفته هستند. و اگر از دامنه کم شد همه اون یال هایی که یک سرش متغیر فعلی هست را اضافه میکنیم بر عکس forward checking که همونجا کار را تمام میکرد. قدرت سازگاری یال تا چه حدی هست یعنی صرفا با بررسی یال میتوانیم بفهمیم که شکست میخوریم واقعا یا نه؟ خوب ما میایم دو تا متغیر را نسبت به همدیگه بررسی میکنیم و به صورت زوج هست حالا اگر 3 تا متغیر را با هم نسبت به همدیگه مقایسه کنیم دیگر این سازگاری یال بدرد ما نمیخورد. پس باید بحث سازگاری را به تعداد بیشتری متغیر نسبت بهم بدیم؟ این معروف هست به اسم سازگاری مسیر: تعاریف سازگاری همگی شبیه به هم هستند در سازگاری یال میگفتیم آیا مقدار سازگاری وجود دارد اگر این مقدار را به متغیر فعلی بدهیم؟ یا میگوییم نسبت به 2 متغیر قابل نسبت دهی هست یا نیست. حالا به جای زوج متغیر، میایم اون زوج متغیر را نسبت به متغیر سوم میسنجیم. اون کم کردن از دامنه هم وجود دارد یعنی بیا از دامنه اون زوج متغیر بیا خط بزن.

یک سازگاری: در سازگاری گره صرفا به محدودیت های یکتایی را در نظر میگیریم یعنی نسبت به خود گره و به طور جدا در صورتی که قبلا نسبت به یکی دیگه مقایسه میکردیم. و شامل یک متغیر هست. در همه گره ها باید باشد اگر میخواهیم سازگار باشد و اون عاملی که باعث نمیشود را حذف میکنیم. همه مقادیر باید محدودیت یکتایی را ارضا کنند. مثلا در رنگ آمیزی نقشه ما محدودیت یکتایی نداریم.

در سازگاری یال میایم یک گره را نسبت به یک گره دیگر مقایسه میکنیم به همین ترتیب k سازگاری تعریف میشود یعنی ما k تا متغیر داریم نسبت به k-1 میتوانیم یک گره را سازگار کنیم یا نه. یعنی واسه 4 سازگاری یعنی بیا یک متغیر را نسبت به 3 سازگاری مقایسه کن. دقت کن به ازای هر انتساب هست. برای رد کردن 3 سازگاری کافی است یک سازگاری 2 تا را پیدا کنیم که قابل تعمیم به 3 سازگاری نباشد. سازگار قوی: علاوه بر k سازگاری باید تا 1 سازگاری هم باشی. تعاریف این 2 فرق میکند. اگر نتوانستی انتساب 2 سازگاری پیدا کنی پس مسئله 3 سازگار هست چون تعریف نقض نمیشود. سازگار قوی بودن مجموعه کامل تری نسبت به k سازگاری هست. اگر یک مسئله سازگار قوی از مرتبه n راس باشد میتوانیم در مرتبه n^2d حل کنیم چون میدانیم حل مسائل CSP با N متغیر از مرتبه D^N هست. چرا ؟ چون فکر کن ما دو متغیر داریم که اولی که میخواهیم مقدار بدهیم 1 سازگار هست. میایم یک مقدار بهش نسبت میدهیم حالا در متغیر دوم باید سراغ مقداری برویم که با اولی سازگار باشد خوب این چه قدر طول میکشد؟ d تا چون کل دامنه رو فقط باید بگردیم. MAC: میاد همزمان که یک متغیر از دامنه یک گره حذف میکند تمام یال های متصل به آن را میاورد و چک میکند. در تست ها دقت کن اگر سازگاری نگفت منظورش همان هیورستیک ها همزمان با forward checking هست. اول هیورستیک MRV بزن بعد که برابر شد دو تا دامنه هیورستیک درجه رو بزن یعنی ببین کدوم بیشترین محدودیت را دارد با گره های مقدار نگرفته. دقت کن AC3 قبل سرچ انجام میشود و سازگاری چک میکند ولی فوروارد چکینگ بعد از اون و در زمان اجرا هست. دقت کن وقتی دنبال همه جواب ها باشیم ترتیب متغیر ها تاثیر میگذارد مثلا در MRV ولی ترتیب مقادیر تاثیری ندارد . دقت کن سازگاری یال پیش پردازش هست و میاد دو به دو چک میکند که به ازای یک مقدار از دامنه دم یک مقدار سازگار در سر داریم یا نه و کمک خاصی به ما نمیکند چرا مثلا در رنگ آمیزی نقشه استرالیا ما با 2 رنگ میتوانیم بدون مشکل سازگاری مسئله را حل کنیم اما اینکار باعث نمیشود که واقعا عملی باشد. سازگاری مسیر قوی تر از یال هست چون دیگه دو به دو نیست چون میایم سازگاری دو متغیر را به متغیر سوم بررسی میکنیم. سازگاری مرتبه 4 میشود که 3 تا متغیر نسبت به متغیر 4 ام یک مقدار سازگار دارد یا نه اگر نداشت مسئله جواب ندارد. دقت کن مسئله میتواند 2 سازگار نباشد ولی 3 سازگار باشد چون شرط سازگاری سوم نقض نشده است که نباشد. ولی 3 سازگار قوی نیست. وقتی سازگار قوی باشد اصلا دیگر نیازی به عقب گرد نداریم. دقت کن که به ازای 1 سازگاری که هیچی به ازای 2 سازگاری میایم و در بدترین حالت d تا مقایسه باید بکنیم چون ممکن است آخرین عدد از دامنه آنها سازگار باشد برای سومی میشود 2 تا d چون دو به دو باید دنبال مقدار سازگار بگردیم.

و این الگوریتم هیچ موقع عقب گرد ندارد چون مطمئنیم یک مقدار سازگار همیشه وجود دارد. محدودیت های عمومی: میتوانیم به محدودیت های دوتایی تبدیل کنیم اما میتوانیم به همون شکلی که بودن روش ساده تری داشته باشیم برای حل کردن یا رد کردن یک سازگاری. اگر m تا متغیر داشتیم و در مجموع دامنه همه آنها n تا مقادیر داشته باشند اگر m>n باشد میفهمیم این محدودیت ارضاپذیر نیست یا سازگار نیست. انتشار باند: دقت کن در مسائل واقعی اندازه دامنه ما خیلی بزرگ هست بخاطر همین دامنه متغیر ها به شکل حد پایین یا بالا هست. و میایم مقادیر را اصلاح میکنیم مثلا در اون مثال هواپیما که دو تا D داشتیم باید مقدار 420 منهای 385 را به عنوان حد پایین اون یکی هواپیما بنویسیم. روش عقبگرد ساده از نوعی که عقبگرد میکند خیلی مناسب نیست چون ما میریم اون گوشه که ربطی به کسی ندارد بررسی میکنیم و عقبگرد میکنیم در عقبگرد پیشرفته باید به سطحی برگردیم که باعث شده یک مقدار از دامنه متغیر فعلی حذف شده است که به این روش back jumping گفته میشود. یعنی میایم به آخرین متغیری بر میگردیم که با متغیر فعلی در تضاد هستند. و به آخرین متغیری که در مجموعه تضاد ها هست عقبگرد میکنیم. پرش به عقب تنها زمانی هست که تمام مقادیر با انتساب های قبلی در تضاد هست یا ناسازگار هست. این روش به نوعی با forward checking در تعارض هست چون اون شاخه ها اصلا هرس میشوند. این ایده بعدا استفاده میشود در این الگوریتم: در الگوریتم قبلی فقط نسبت به یک متغیر بود و ممکن بود متغیر مقادیر سازگار را داشته باشد ولی back jumping متوجه نمیشود که این شاخه به طور کلی بعدا به مشکل میخورد. چون فقط به یک متغیر دقت میکرد. در واقع اون متغیر فعلی با مجموعه متغیر های بعدی باعث این میشود که در نهایت باعث ناسازگاری میشود که این الگوریتم میگوید همچنان اون پرش را انجام بده که نام direct back jumping نام دارد. مجموعه تضاد مثل قبل هست برای گره های قبلی ولی وقتی که یک پرش را انجام میدهیم اون موقع میایم مجموعه تضاد فعلی را به مجموعه تضاد گره ای که بهش پرش کردیم اپدیت میکنیم مگر اینکه یکی باشند که اضافه نمیکنیم چون معنی ندارد. یعنی گره بالایی به پایین تر از خودش را هم نگاه میکند. مسئله atmost بیا min ها را جمع بزن در دامنه ها اگر بیشتر از منابع شد صد در صد میگیم مسئله جواب ندارد. ماکس ها را هم میتوانی خط بزنی چون نگاه حداقل 2 هست اگر به یکی بدهی 5 به 3 تای دیگه نمیتوانی 2 تا بدهی بیشتر از 10 تا میشود و مسئله جواب ندارد همینطور برای 6 ولی برای 4 مشکلی ندارد. 35 چطوری بدست اومد؟ 420 را منهای حد بالای اون یکی که 385 هست کردیم کلا تو اینجور مسائل با حد بالا و پایین سعی کن بازی کنی دقت کن به ازای مقادیر زیر 35 همچین چیزی وجود ندارد چون 420 تا مسافر نمیشود. 255 به همین شکل بدست آمده است. دقت کن تو رنگ امیزی محدودیت تکی نداریم مگر اینکه خودش بگوید که یک گره دلش نمیخواهد سبز شود اون موقع دامنه خالی میشد و باید عقب گرد میکردیم برای گره D مثلا در تست سال 89. تناقض یعنی اینکه کدام روش باعث خالی شدن دامنه میشود.

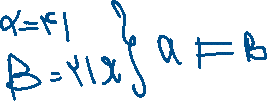
کلا جستجو محلی با فرموله سازی افزایشی کار نمیکردند و با فرموله سازی حالت کامل کار میکردند که یک حالت مسئله باید دقیقا یک مقدار دهی به همه متغیر ها باشد. حالا در CSP میخواهیم بررسی کنیم: mic-conflicts: اون حالتی رو بردار که تعداد محدودیت های ارضا نشده را کمینه کند. حالا چطور میتوانیم از ساختار گراف محدودیت مسئله استفاده کنیم تا یک مسئله CSP را راحت تر و سریع تر حل کنیم: اولین ایده این هست که زیر مسائل کاملا مستقل را جداگانه حل کنیم بعد با هم ادغام کنیم و با این روش الگوریتم از مرتبه خطی بر حسب N میشود به جای اینکه نمایی باشد. اگر گراف محدودیت به شکل درخت باشد یعنی دور نداشته باشد و بین دو گره دقیقا یک مسیر باشد میتوانیم از مرتبه خطی nd^2 حل کنیم. این الگوریتم یک درخت را به عنوان گراف محدودیت ورودی را میگیرد و باهاش یک حالتی را پیدا میکند که سازگار باشد در آن همه محدودیت ها. اول درخت را ریشه دار میکنیم به صورت دلخواه و بعد باید تمام یال هایی که به اون متصل هستند را خارج کنیم بعد برو برای گره بعدی همین کار را بکن. در واقع یال ها باید از پدر به فرزندان باشد. مرحله دوم باید سازگار کنیم باید از برگ ها شروع تا به ریشه برسیم. اینجا هاش باید از دامنه حذف کنی. از مرتبه زمانی دقت کن چون داریم n-1 تا را سازگار میکنیم.

این فرمول را بقیه جاها میتوانیم استفاده به گونه ای که یک گره را مقدار دهی و بعد حذف کنیم و بعد دامنه سایر را اپدیت کنیم بعد با همون چیزی که بلد بودیم با همون مرتب زمانی حل کنیم. یعنی یکبار آبی بدهیم حذف کنیم حل کنیم و ... این حذف کردن باید باعث تشکیل یک درخت شود و تو زمان خوبی حل میکنیم. D^C چی بود؟ تعداد راس ها و طبیعتا به جای ND^2 اینجا N-C داریم ضریب D^C چی بود چون ما تو بدترین حالت به ازای هر انتساب یکبار باید کل درخت را حل کنیم. و این بستگی به گراف دارد که چه قدر شبیه به درخت هست حالا اگر گراف کامل باشد یعنی بین هر دو راس یک یال داشته باشیم اونموقع برای تبدیل به درخت باید همه راس ها جز 2 تا را حذف کنیم تا درخت شود چون اگر 3 تا باشد تشکیل دور میدهد. و عملا به ما کمک خاصی نمیکند. پیدا کردن اون C سخت هست. یک روش دیگر این هست که بیایم یک سری ابر گره بسازیم که هر کدوم نماینده یک زیر درخت باشند و روی هر کدوم از این ابر گروه ها بیایم الگوریتم مورد نظر را حل کنیم. دامنه ابر گره ها یک دامنه سازگار از اون مسئله هست. یعنی خودش یک انتساب سازگار از گره های داخلش هست یعنی انتسابی که میدهیم جز اون M1 هستند مثلا. محدودیت بین ابر گره ها این هست که گره های مشترک مقدار یکسان حتما باید داشته باشند. دقت کن همه گره ها باید رسیدگی شود و اگر محدودیتی بین دو تا متغیر داشته باشد حتما باید جز یکی از ابر گره ها باشد. دقت کن اگر یک گره در M1 بود اون یکی در M4 باید بین این ها هم اون گره ظاهر شده باشد. روش حل مسئله: اولا همه زیر مسائل به طور مستقل حل شود. اگر یکی جواب نداشت مسئله اصلی جواب ندارد. اگر یکی جواب داشته دامنه آن همه جواب های زیر مسئله هست. محدودیت میشود اون متغیر های مشترک. تحلیل این روش: عرض درختی اندازه بزرگترین زیر مسئله منهای یک یعنی اون ابر گره ها اندازه بزرگترین آنها 3 هست پس عرض میشود 2. عرض درختی یک گراف: مینیموم عرض زیر درخت هاش اونموقع مرتبه میشود d^w+1 +n که از مرتبه خطی هست.

تقارن مقادیر: تو بخش قبل از گراف کمک گرفتیم تا حل کنیم اما حتی مقادیری که تو دامنه گره ها هست خوب هستند و میتوان از روش های دیگر استفاده کرد حالا منظور از این خوب بودن چی هست؟ چه ویژگی دارد ؟ یعنی ما بتوانیم از جواب فعلی سایر جواب ها را بدست بیاوریم. چون بین مقادیر متغیر ها هیچ فرقی با هم ندارند و یک تقارن هستند و یک جایگشت دیگر بدست بیاوریم چند حالت میشود 3! چون 3 رنگ هست داخل دامنه متغیر ها و 6 تا جواب میتوانیم بسازیم. اگر از این ویژگی استفاده نکنیم و از عقبگرد استفاده کنیم درخت خیلی بزرگ میشود. راه دیگر این هست که بیایم تقارن درخت را بشکنیم تا یک جواب بدست بیاوریم تا از روی اون بقیه جواب ها را بدست بیاوریم. مثلا بگیم آبی ارزش بیشتری نسبت به قرمز و سبز دارد خوب ما بعد میگوییم که مثلا فلان گره ها باید رنگ پر ارزش تری بگیرند پس فقط تو یک حالت به جواب میرسیم و بعد میتوانیم تعمیم بدهیم. چون دقت کن یک محدودیت جدید را اضافه کردیم و باید سازگاری را چک کنیم.

**فصل هفتم بخش اول:**

دیگه کلا با عامل های منطقی کار داریم. حالا ببینیم منطق چیست؟ مشکل عامل های قبلی این بود که دانش از محیط خیلی کمی داشت توی ارضا محدودیت یکم این درک را از حالت بیشتر کردیم ولی بازم نیاز به بیشتر داریم ولی میخواهیم عامل هایی بسازیم که در هر لحظه یک سری دانشی در پایگاه دانش خود دارد و علاوه بر این یک سری اطلاعات از حالت فعلی هم دارد عین انسان، و با توجه به این 2 میتوانیم استنتاج کنیم پس خوبی این عامل ها این هست که میتوانند دانش را ذخیره کنند و استنتاج کنند. منطق یعنی اینکه ما دانش خود را هم بتوانیم اطلاعات عامل را نشان دهیم هم امکان استنتاج را برایش فراهم کنیم. به چیزی که امکان این استنتاج را میدهد منطق میگوییم. مثلا در مسئله هشت وزیر نمیتوانست استنتاج کند که دو عدد روی یک کاشی نمیروند. پایگاه دانش همونجایی هست که همه اطلاعات آنجا هستند. با استفاده از تابع TELL دانشی اضافه میکنیم به پایگاه دانش عامل و با ASK از پایگاه دانش سوال میکنیم مثلا میگوییم آیا X بر 2 بخش پذیر هست بعد از عامل میپرسیم که X میتواند فرد باشد؟ اینگونه استنتاج میکند هر عدد یا فرد هست یا زوج پس چون بخش پذیر بر 2 هست پس زوج هست. در واقع اینکه چه اکشنی باید انجام بدهیم را از پایگاه دانش میپرسد. در واقع خروجی ASK همان اکشن هست. این برنامه در سطح دانایی وجود دارد یعنی مثل قبلی ها نیست که یک درکی بگیرد و اکشن خروجی بدهد این بر اساس سطح دانش نتیجه گیری میکند مثلا اینکه از یک خیابون مشخص باید حرکت کنیم. دو روش داریم روش اعلانی این هست که اول همه دانش را بدهیم بعد فقط بیاد استدلال را انجام بدهد روش دوم یا رویه ای به این شکل هست که استدلال را بدهیم ولی دانشی ندهیم. **ومپوس**: درکی که داریم از یک محیط این هست که اگر نزدیک اون ومپوس باشیم یک بویی حس میکنیم از اطراف یک درک دیگر این هست که اگر اطراف چاله باشد یک نسیمی میاد خونه های اطراف یعنی. اگر دیوار باشد که درک ضربه رو میگیریم انگار برخورد کردیم و اگر اون ومپوس را بکشیم، صدای جیغ از محیط دریافت میکنیم. 1 تیر هم بیشتر نداریم. محیط ما گسسته هست و اتفاقا تک عاملی هستند چون ومپوس عامل جدا نیست و کاری قرار نیست انجام بدهد تا معیار کارایی ما عوض شود و محیط مشاهده پذیر جزئی هست. پس منطق یک زبان رسمی برای بازنمایی اطلاعات و دانش هست و البته که میتواند روی آن دانش استنتاج و نتیجه گیری بکنیم. هر پایگاه دانشی از یک سری جملات تشکیل شده است که باید منطبق بر syntax اون زبان باشند یعنی مثلا x=5 که خوب این درست هست ولی مثلا 5=x جمله درستی نیست. و یک زبان باید معنا را هم مشخص کند که semantic نام دارد یعنی این جمله درست هست یا نه. مدل همون دنیای هست که اون زبان منطقی توش هست منتها انتزاعی هست. ارزش یک جمله را بر اساس دنیایی که داخلش هستیم میسنجیم. اسم اون دنیا مدل هست. به دنیای ممکن مدل میگوییم مثلا x انتسابی از پاهای انسان در دنیا هست فقط به شکل انتزاعی شده هست. M(a) یعنی همه مدل های الفا وقتی یک جمله در یک مدل باشد میگوییم m یک مدل a است. استلزام: جمله الفا جمله بتا را استلزام یا ایجاد میکند اگر هر وقت الفا درست باشد بتا هم درست هست. مدل یک انتسابی هست از متغیر های داخل یا اسامی داخل جمله به مقادیر و قوانین و ارزش های این مدل بر اساس همان ارزش های ریاضی هست یعنی 2+2 میشود 4. یا اینطوری بگیم یک مدل انتسابی هست از اسامی داخل یک جمله به ابجکت های درون اون جهان.



یعنی از درستی الفا میتوانیم درستی بتا را نتیجه بگیریم. در واقع بتا نتیجه منطقی الفا هست. پس الفا زیر مجموعه بتا میشود چون میخواهیم بگیم هر جا الفا درست بود بتا هم درست هست و برعکس آن درست نباشد یعنی جاهایی که بتا درست هست لزومی ندارد الفا درست باشد. دقت کن الفا میدهد بتا را یعنی الفا قوی تر هست و فرض قوی تری را میدهد یعنی اینکه x مضرب 4 هست اطلاعات بیشتری هست از اینکه بگیم x زوج هست. الفا درست بود بتا باید درست باشد.

یک الگوریتم استنتاج sound یا صحیح میاد تمام استنتاج هایی را میدهد که واقعا توسط KB استلزام میشود یعنی خارج از اون را نمیدهد. کامل بودن هم یعنی هر جمله که توسط KB میشود را آن الگوریتم هم بتواند. و واقعا هم بتواند نتیجه بگیرد. دقت کن کل KB را میتوانیم در قالب یک جمله با AND بین آنها نمایش دهیم.

حالا از این اطلاعات جملات مختلف چی استلزام میشوند؟ یعنی از KB ما چه نتایجی را میتوانیم بدست بیاوریم یا استنتاج کنیم، استنتاج کردن یعنی از پایگاه دانش ما چه جملاتی استنباط میشوند؟ دقت کن به ازای هر خانه در پایه 2 حالت داریم یعنی مثلا اگر 3 خونه هست هشت حالت داریم که مثلا چاله هستند یا نیستند. مثلا در خونه 2,1 نمیتواند چاله باشد چون ما درک میکردیم که بادی هست پس اون حالت درست نیست. حالا باید ببینیم الفا 1 ما توی اون خونه چاله هست یا نه. یعنی نتیجه بگیریم که در خونه 1,2 چاله هست یا نیست اگر بتوان نتیجه گرفت که درست هست اگر نباشد که هیچی. مجموعه مدل های پایگاه دانش بر اساس درک ما از محیط ساخته میشود مثلا در خونه 2,1 ما نسیم احساس میکنیم پس میدانیم در خونه بالایی یا راستی چاله هست یا نیست. دقت کن که تمام اطلاعات KB را میتوانی در یک جمله بنویسی. KB میشود قوانین ومپوس بعلاوه درک هایی که من از محیط گرفتم. دقت کن بر اساس اون جمله الفا باید تمام KB را پوشش دهد یعنی KB درست هست الفا هم اونجا ها درست هست یا نیست. برعکس مهم نیست یعنی KB غلط باشد الفا درست باشد برای ما مهم نیست. به فرآیند جواب دادن به این میگویند استنتاج. یعنی یک KB میدهیم و یک جمله که بفهمیم جمله را میتواند استنتاج کند یا نه. یک الگوریتم صحیح است یا sound یعنی خروجی واقعا درست باشد یعنی واقعا بتوان توسط KB جمله مورد نظر استلزام شود. کامل هست اگر هر جمله درون KB توسط الگوریتم ها استنتاج شود. اگر KB غلط باشد هر جمله ای را میتوان استلزام کرد چرا؟ چون تهی هست. حالا چطور بدانیم KB درست هست؟ خوب این عامل از طریق سنسور ها درک را دریافت میکند و قوانین کلی با متد های یادگیری در طول زمان بدست آورده است یعنی بر اساس تجربه. **منطق گزاره ای**: اول باید سینتکس را بررسی کنیم یعنی شکل جملات چه شکلی هست بعد سمنتیک که آیا جملات درست هستند یا نه. سینتکس: یک سری جملات داریم که شامل یک نماد گزاره ای هستند و هر نماد گزاره ای دارای کلا دو مقدار هست صحیح یا غلط. دو گزاره داریم که معنی آن از قبل مشخص هست T همیشه درست F همیشه غلط. جملات غیر اتمی از ترکیب یک سری جملات اتمی و نماد ها تشکیل میشوند به یک نماد گزاره ای لیترال گفته میشود مثلا NOT P یا P. AND: ترکیب عطفی. فصلی: OR. ایجاد: شرطی. با این 4 علامت میتوان منطق گزاره ای را ساخت. و با استفاده از گرامر هر جمله ای را میسازیم. اولویت ها را هم حفظ کن. در این منطق کلمه North با کلمه N هیچ فرقی ندارد این شمال بودن را ما میفهمیم نه آن زبان. در منطق گزاره فقط True و False داریم از لحاظ معنا و نماد و ابجکت در جهان ما هستند یعنی همه چیز یا درست هست یا غلط و بر اساس اون نحو میگیم معنای این جمله چیست. دقت کن دید مجموعه ای داشته باشد چون هر کدام از حالات این نماد ها به ازای مقادیری خاص درست هستند نه همه مقادیر و الفا همیشه اتمیک نیست بلکه ممکن هست مجموعه از جملات باشد. نات میگذاریم یعنی وجود ندارد مثلا.

حالا قوانین یک زبان که ببینیم این جملات درست هستند یا نیستند یک انتساب صحیح یا غلط به نماد های اون جمله این میشود مدل اون جمله. وقتی نات میگذاریم یعنی نسبت به اون ادراک چیزی دریافت نکرده ایم پس نیست. کلا میخواهیم بدونیم ما بر اساس KB میتوانیم الفا را نتیجه بگیریم یا نه اینکه بیایم دونه به دونه در جدول صحبت بشمریم زیاد مناسب نیست چرا چون اگر ما N نماد داشته باشیم 2^N حالت میشود و زمان نمایی میشود و مجبوریم روش های دیگه ای معرفی کنیم. ولید: جمله ای درست هست که در همه مدل ها درست باشد همون تاتولوژی هست یعنی 2^N حالت داشتیم توی تمام حالات درست باشد. دقت کن شرطی در یک مدل خاص هست ولی قضیه استنتاج در همه مدل ها منظورش هست. یعنی تو هر مدل که الفا درست باشد بتا هم درست هست پس تو هر مدل الفا میدهد بتا درست هست و این یک تاتولوژی یا ولید بودن هست این انگار همان تعریف تاتولوژی است. ببین یک روش این بود که مدل ها را در جدول درستی بشمریم یک حالت هم این بود که بریم بر اساس قوانین اثبات کنیم به روش دومی همین میگویند استنتاج.

روش استنتاج به این شکل هست که از اون 2N حالات ممکن ما بیایم چک کنیم به ازای هر حالتی که KB درست هست a هم درست باشد. و اگر حالتی باشد که KB درست باشد ولی a درست نباشد جواب آن no هست. پس مثل دنیای ومپوس اول بیا ببین KB کجا ها درست هست مثلا 3 حالت شده است به ازای تمام 128 حالت، بعد بیا ببین a هم آنجا درست هست یا نه که خوب به ازای اون 3 حالت درست بود هست چون خود جمله P1,2، false بوده است و ~P1,2، True میشود. و 125 تا بعدی برای ما مهم نیست. اگر یکی از این 3 حالت a در آن درست نبود میگوییم استلزام نمیکند الفا را.

مفاهیم منطقی: هم ارزی: هر جا این درست بود اون یکی هم درست باشد و برعکس. نگاه ما به جملات مجموعه ای هست. valid: جمله ای که به ازای تمام 2N حالت درست هست که تاتولوژی هم گفته میشود. دقت کن A entail B فرق دارد با A=>B. دومی بر اساس مدل هست همان اسامی هستند که به یک سری متغیر منتسب شده اند ولی اول جوابش کلا بله/خیر هست. اما ارتباط بین این 2 چیست؟

**استنتاج**: اگر الفا entail کرد بتا را آنگاه a=>b. کلا هر جا نات پی با خود پی or بود بدون True هست و اگر and بود بدان False هست. دقت کن الفا و بتا یک چیز نیستند ولی با هم در ارتباط هستند. دقت کن مجموعه مدل هایی که False در آنها ارزش درست داشته باشد تهی هست و تهی زیر مجموعه همه مجموعه ها هست پس میتوانیم بگیم فالس entail میکند الفا را همیشه درست هست.

**ارضا پذیری**: جمله الفا ارضا پذیر است که در بعضی یا حداقل یکی از مدل ها درست باشد همه ولید ها ارضا پذیر هستند که خوب طبیعی هست اصلا. ولید یعنی یک مدلی هست که به ازای آن ارزش FALSE داشته باشد جمله ما پس فقط کافی هست یک جا را پیدا کنی تا ارزش false داشته باشد آنگاه دیگر تاتولوژی و ولید نیست. و اگر جمله ما به شکل p=>q بود و ما به ازای حالت مقدم درست تالی غلط نتوانیم حالتی پیدا کنیم پس اون جمله ولید هست چون فقط توی این مدل از تالی و مقدم اون عبارت ارزش false دارد. دقت کن به ازای y or not x یک جمله ولید نیست چون مدلی وجود دارد که در آن این جمله ارزش False دارد. اگر بخواهیم ثابت کنیم که از الفا بتا را نتیجه بگیریم یا استنتاج کنیم. میتوانیم اینطوری اثبات کنیم که الفا اند نات بتا ارضا ناپذیر هست و براش الگوریتم هم معرفی میکنیم. ارضا پذیر نباشد یعنی همیشه غلط هست. الگوریتم رزولوشن میگوید فقط من رو اجرا کن جای این استنتاج ها. یکنوایی: اگر KB بدهد الفا را اونموقع KB AND B هم میدهد الفا را یعنی با اضافه شدن جملات جدید قدرت بیشتر میشود نه کمتر. دقت کن هر موقع گفتند جملات منطق گزاره ای خود اون نماد میتواند جملات پیچیده ای باشد پس دیگر 8 تا حالت انتساب نیست. کلا به این جملات در منطق گزاره ای دید همان منطق سابق را نداشته باش و دید مجموعه ای داشته باش. تست3: دقت کن چون الفا entail نمیکند گاما را پس اولا false نیست پس ارضا پذیر هست حتما طبق قاعده بالا که false هر چیزی را entail میکند و بعد اگر b و گاما ارضا ناپذیر باشند یعنی هر دو تو همه مدل ها ارضا ناپذیر بوده اند پس false هست ولی گفته که الف entail نمیکند گاما را و میدانیم فقط false، entail میکند false را پس الف باید false باشد که نمیشود پس این کلا غلط هست و دقت کن اگر الف entail نمیکند گاما را به معنای ارضا ناپذیر بودن نیست به معنای ولید نبودن هست چون یک مدلی پیدا کردیم که به ازای آن True نیست. پایان تست 3 ادامه را در ادامه جمله قبل تست 3 بخوان. یعنی اون دانشی که اضافه میکنیم ضربه نمیزند به استنتاج جدیدی. حتی اگر نقض الفا باشد اون جمله ای که اضافه میکنیم موردی ندارد چون اون KB AND B میشود غلط و غلط هر چیزی را میدهد.

**رزولوشن**: الگوریتم های قبلی چون کامل نبودند اون قوانین استنتاجی همگی sound بودند ولی کامل نبودند پس نمیتوانستیم به صورت الگوریتم بگیم چون کامل نبودند و نمیشد داخل الگوریتم از آنها اضافه کرد. روی کلاز ها کار میکند یعنی or یک سری لیترال. دقت کن جفت به جفت حذف میکنی کلا حذف نمیکنی تا بشود تهی. نیاز به الگوریتم رزولوشن داریم که با سرچ ترکیب شود و به ما نشان بدهد که آیا از KB هر آلفایی را میتوانیم نتیجه بگیریم یا نه. فقط دقت کن در رزولوشن در هر جمله یک لیترال فقط یکبار باید حضور پیدا بکند. کلاز ترکیب فصلی لیترال ها هست و فقط روی کلاز ها اعمال میشود. این قانون پایه ای برای بقیه الگوریتم ها هستند. پس باید به صورت CNF یا اند یک سری عبارت ها بدست بیاوریم یا اند اور ها. و هر پایگاه دانشی را میتوان به فرم CNF بدست آورد. یعنی and یک سری کلاز میشود CNF که داخل هر کلاز یک سری لیترال با OR ترکیب شده اند. هر جمله منطق گزاره ای یک تیپ به فرم CNF دارد. مراحل تبدیل به CNF حفظ شود. کلاز بود اور یک سری لیترال CNF اند یک سری کلاز هست. الگوریتم رزولوشن: میاد برای KB بدهد الفا نشان میدهد که نات الفا هست و KB درست هست ارضا ناپذیر هست. و اینقدر این الگوریتم که همین حذف کردن رزولوشن هست اعمال میکنیم تا به تهی برسیم به تهی رسیدیم یعنی اینکه KB میدهد الفا چون به تهی برسیم یعنی به تناقض رسیدیم ولی اگر به تهی نرسیم و کاری نتوانیم بکنیم یعنی KB نمیدهد آلفا را. الگوریتم رزولوشن کامل هست ولی خود رزولوشن کامل نیست. دقت کن که عمل فکتورینگ باعث میشود لیترال ها و طول عبارات محدود بماند. دقت کن که باید با گزینه ای که میخواهیم درستی را نتیجه بگیریم نقض را اضافه کنیم بعد به تهی برسیم. دقت کن تک لیترال یک عبارت اور هست. وقتی میگوییم رزولوشن زدیم یعنی دست کم یک لیترال بوده که توی اون یکی عبارت نقیضش بوده که اعمال کردیم. همیشه دو عبارتی که لیترال متناقض هم دارد یعنی یکی لیترال نات یکی لیترال درست داشت را با هم اجتماع بگیریم یا اور کنیم همیشه درست هست چون اون 2 لیترال را کنار هم میاوریم و کل اور درست میشود. رزولوشن میگوید اگر الفا درست باشد اگر بتا درست باشد اون موقع حاصل رزولوشن این 2 درست هست این برعکس آن درست نیست رزولوشن با اند یکی نیست.

**فصل هفتم بخش دوم:**

الگوریتم رزولوشن الگوریتم کاملی هست البته یک سری جملات خیلی زیبا هستند که راحت تر میشود حل کرد یکی مثلا میشود عبارات معین که همین ترکیب اور لیترال ها هست ولی دقیقا یک مثبت دارد definite clause نام دارد. دقت کن که الگوریتم رزولوشن کامل هست و روی جملات قابل استنتاج جواب بله و روی جملات غیر قابل استنتاج جواب نه را میدهند. تمام شدن قابل اثبات هست ولی دقت کن غیر قابل استنتاج بودن لزوما تمام نمیشود. فرم هورن هم هست که همون ترکیب فصلی لیترال ها هست ولی حداکثر یک لیترال مثبت دارند یعنی میتوانند لیترال مثبتی هم نداشته باشند. و یک دسته دیگر کلاز های هدف هستند که همون ترکیب فصلی هستند منتها لیترال مثبتی ندارند. پس جملات هورن یا definite clause هستند یا کلاز هدف. هورن تحت رزولوشن بسته هست یعنی دو تا رو با هم رزولوشن بزنی چون یک مثبت را با یک منفی میزنی حاصل حداکثر یک لیترال مثبت دارد و حاصل هم هورن میشود. خوبی پایگاه دانش هایی که به این شکل هستند یک سری ویژگی دارند مهم ترین ویژگی این هست که میتواند به فرم implication یا شرطی نوشته شود. و یک سری روش ها داریم به نام زنجیره سازی رو به جلو و عقب که در پایگاه دانش های مخصوص به فرم عبارات معین انجام میشوند. به یک تک لیترال مثبت یک فکت گفته میشود. تک لیترال را به صورت true=>p میتوانیم نشان دهیم. در زمان خطی قابل حل هستند برخلاف رزولوشن که در زمان نمایی هست. این هم به همین شکل هست که آیا KB استلزام میکند الفا را البته باید به شکل شرطی باشد دیگه. از فکت ها شروع میکنیم که در KB هستند و اگر مقدم برقرار بود اونوقت تالی هم اضافه میکنیم به فکت ها اینقدر ادامه میدهیم تا نشود فکتی اضافه کرد یا به الفا نرسیم جواب نه میدهیم و یا به الفا میرسیم. عبارات معین همان شرطی ها هستند. یعنی با این فکت ها ببین چپ کدوم برقرار هست پس راست را اضافه کند عین MP هست. و بعد میایم هر جا که اون تک لیترال ها وجود دارند یک عدد کم میکنیم از ریشه تلاقی این 2. اونها که عددشون صفر شود یعنی تمام پیش شرط ها ارضا شده اند و بعد هر چیزی که اضافه شد میریم از بقیه هم کم میکنیم تا به صفر برسد. به این میگویند forward chaining. این الگوریتم صحیح هست چون بر اساس MP هست و روی عبارات معین کامل هست. این الگوریتم ها مبتنی بر داده هست و از داده KB استفاده میکند به اصطلاح data-driven هستند. الگوریتم زنجیره سازی رو به عقب دقیقا برعکس این شرایط هست یعنی از هدف میرسیم به تک لیترال ها برای اثبات تالی. یعنی از تالی به مقدم حرکت میکنیم. حالت بازگشتی دارد یعنی هعی از تالی به مقدم حرکت میکنیم. دقت کن از فکت ها هم میتوانیم استفاده کنیم. این الگوریتم مبتنی بر هدف هست. و مرتبه زمانی بسیار کمتری از هدف دارد. SAT مسئله ارضا پذیری هست که یک CNF را به عنوان ورودی میگیرد همون اند یک سری کلاز یا اند اور ها و باید تعیین کنیم این جمله ارضا پذیر هست یا نه مثلا یک انتساب صحیحی به کلاز ها میتوانیم بدهیم. این یک مسئله NP هست و بهتر از زمان نمایی برای حلش نداریم. برای اینکه بفهمیم الفا استلزام میکند بتا را باید بفهمیم که الفا اند نات بتا ارضا نا پذیر هست یا این الگوریتم SAT بهش جواب نه برمیگرداند. اولین الگوریتم DPLL هست که میاد به صورت عمقی یک CNF را میگیرد تا ببیند ارضا پذیر هست یا نیست. یعنی هعی تک لیترال ها را true میدهد تا ببیند کدوم کلاز ها ترو میشوند. البته یک سری حقه میزنیم که سریعتر به جواب برسیم. اولیش early termination هست که همینطوری میرود پایین چک میکند که کلاز ها ارضا شده اند یا نه. یعنی درستی یک کلاز را زودتر تشخیص میدهد. از اینکه استفاده میکند که بین چند تا اور اگر یکی ترو شود کل ترو میشود و دیگر لازم نیست بقیه چک شود. و مثلا تا یک جایی میرود متوجه میشود این شاخه دیگر به ارضا نمیرسد و میرود یک شاخه دیگر. غلط بودن کل عبارت رو اینطوری تشخیص میدهد که یکی از کلاز ها غلط شده باشد و درست بودن را به این شکل تشخیص میدهد که یک لیترال بین اور ها درست شده باشد در واقع زودتر به جواب میرسد. مورد بعدی مشخص کردن نماد های خالص هست یعنی لیترال هایی که در همه کلاز ها یا مثبت آمده است یا منفی. این سمبل های خالص یک خوبی دارند این هست که میایم به نماد های خالص ترو میدهیم تا چک کنیم درست هست یا نه حتی لازم نیست غلط بدهیم چون میدانیم اتفاقی نمیفتد و اون شاخه رو دیگر چک نمیکند. ببین کلا توی این DPLL میخواهیم یک مدل پیدا کنیم که ارضا پذیر و درست باشند. یک سری کوتاه کردن هم میتوانیم داشته باشیم مثلا اگر بتا را فالس بدهیم کل عبارت ترو شود خوب اون رو کلا حذف میکنیم. یا مثلا به لیترالی که کلا یکبار ظاهر شده است یک مقدار درست میدهیم. مورد آخر unit clause هست که کلا یک لیترال مثبت یا منفی دارد. و باید به اینا زودتر مقدار دهی کنیم و یک جوری هم مقدار بدهیم که مثبت شوند. یعنی مثلا اگر نات هست فالس میدهیم بعد درخت را چک میکنیم اگر با نات بود فالس اگر بدون نات بود ترو باید بدهیم. مقدار دهی ما به یک عبارت واحد میتواند عبارت های واحد دیگری تولید کند. و روی عبارت های معین مثل forward chaining عمل میکند. در DPLL اول به یونیت کلاز ها مقدار بده. یونیت کلاز بدست بیار ترو بده. دقت کن در یونیت کلاز باید جوری مقدار بدهیم که حتما مثبت بشوند پس اگر ترو دادیم تناقض شد یعنی غلط هست و درست نیست. دقت کن کلا داری شاخه های درخت را هرس میکنی. در این فصل هم میتوانیم از جستجو های محلی استفاده کنیم پس باید از فرموله سازی کامل استفاده کنیم پس میتوانیم باهاش مسئله SAT را حل کنیم. یعنی با لوکال سرچ دنبال یک مدلی بگردیم که جمله داده شده را ارضا بکند. فضای حالت: انتساب کامل فرموله سازی کامل. تابع هدف تعداد کلاز های ارضا نشده باشد. اکشن ها که همسایه رو تولید کنیم و بهش حرکت کنیم مثلا یک لیترال را انتخاب میکنیم اگر در حالت فعلی ترو باشد فالس میکنیم و برعکس با این کار همسایه تولید میشود. چون خیلی تپه های محلی داریم باید از تصادفی بودن استفاده میکنیم و اسم الگوریتمش WALSAT که همان رویکرد حریصانه و تصادفی را دارد و یک انتساب تصادفی میدهد اگر همه رو ارضا کرد که هیچی. اگر یک سری کلاز ماندند که ارضا نشدند به صورت رندوم انتخاب میکنیم و با استفاده از 2 روش میگوییم که مقدار کدوم لیترال را عوض کند. روش اول میگه اونی را انتخاب کن که مقدار کلاز های ارضا نشده را کمینه کند یعنی نه تنها کلاز خودش را ارضا کند بقیه رو هم ارضا میکند. روش دوم: یک لیترال را رندوم انتخاب میکند و عوض میکند تا شانسی یکی خوب شود. روش اول همون حریصانه هست دنبال خوب شدن روش دوم همون تصادفی هست. به احتمال P یک کلاز را انتخاب کن به صورت تصادفی و به احتمال 1-P بیشترین کلاز را ارضا کند. اگر max flip را بینهایت بگذاریم اینقدر میگردد تا اون مدل را پیدا کند اگر باشد ولی اگر نباشد الگوریتم در بینهایت گیر میکند. یعنی ارضا ناپذیری را همیشه تشخیص نمیدهد و خوب دنبال ارضا ناپذیری بودیم در استنتاج ولی از این الگوریتم WAKSAT نمیتوانیم استفاده کنیم.

سختی مسائل SAT: این مسائل NP COMPLETE هستند یعنی الگوریتمی وجود ندارد که همه مسائل این حوزه را در زمان کمتر از نمایی حل کند. یک نسخه سخت یا آسون هست. بعضی ها خیلی اسون مثل 2-SAT یعنی کلاز های ما 2 تایی باشد یعنی 2 لیترال داشته باشند و راه حل چند جمله ای دارند ولی 3-SAT این شکلی نیست. مسائلی که راه حل ها به طور یکنواخت در فضای مسئله توزیع شده اند برای جستجو های محلی ساده ترند مثل N وزیر که با یکی دو گام به هدف میرسیم. البته به صورت میانگین. مسائل سخت: تعداد کلاز ها نسبت به متغیر ها زیاد هستند یعنی نسبت کلاز به متغیر زیاد میشود و سخت تر میشود. حالا برمیگردیم سراغ ومپوس: عامل مبتنی بر دانش هر اکشنی که انجام میدهد را به KB اضافه میکند. یک سری اطلاعات پایه هم داریم مثلا 1.1 هیچ چاله ای نیست و دقت کن برای هر 16 تا خونه بنویسیم. یعنی هر خونه یک سری قوانین دارد و طبق منطق گزاره ای باید بنویسیم. با انتخاب 2 از 16 باید بنویسیم که در کدام خانه ها ومپوس نیست. باید ادراک هم اضافه کنیم به پایگاه دانش. دقت کن ادراک بر حسب زمان هستند پس باید یک برچسب به ادراک بزنیم که با همدیگر قاطی نشوند اسامی یکسان مثل بو در بازی ومپوس. کلا یک پارامتر زمانی نسبت به هر چیزی که تغییر میکند در نظر بگیر. به هر جنبه که تغییر میکند میگوییم فلاند و اونایی که ثابت هستند میگوییم temporary. میتوانیم ادراک را به ویژگی های هر خانه مرتبط کنیم یعنی مثلا بگیم هر موقع تو خونه فلان شماره قرار گرفتی مثلا یک نسیمی حس کردی. به صورت یک عبارت دو شرطی میتوانی بنویسی که جفت طرف باید برقرار باشند. کلا در منطق گزاره ای متغیر نداریم پس واسه هر 16 خونه باید بنویسیم. بعد باید بیایم یک سری نماد اضافه کنیم بعد باید یک سری جملات به نام affect Asim بنویسیم که نشون بدهیم نتیجه اکشن در جهان چی بوده است. مثلا اگر در ثانیه صفر هستیم از خونه دو بریم چپ مثلا نتیجش میشود این. دقت کن کلا اونهایی رو ثبت میکنیم که مدام تغییر میکنند ولی اونایی که بعد از اکشن ثابت میمانند KB خوب نمیتواند استفاده کند و به این مشکل قاب میگویند مثلا در خونه راست رفتی ربطی ندارد تیر تموم شود یا باهات بیاد. یک راه اول این هست که بیایم یک شرطی بدهیم مثلا رفتیم راست اگر تو زمان فعلی تیر داشتی تو زمان بعدی هم تیر داری دو شرطی. اینطوری حفظ میکنیم. و از مرتبه O MN هست که زیاد جالب نیست راه بهتر این هست به جای نوشتن قانون برای اکشن بیایم قانون بنویسیم واسه اون فلوئنت ها یا ثابت ها یعنی بگیم خوب تو تیر داری تو ثانیه بعدی اگر و فقط اگر تیر داشتی و شوت نکردی. یعنی اکشنی که تو زمان بعدی انجام دادیم باعث تغییری در آن نشده است. حالا که همه فکت ها و همه قوانین در KB هستند و ادراک را هم درون KB میگذاریم و عامل هر چیزی از پایگاه دانش بپرسد KB میبیند با استفاده از یکی الگوریتم هایی که آموختیم میبینید که الفا که سوال عامل هست را میدهد یا نه به این شکل عامل هر لحظه میتواند استنتاج کند مثلا در ثانیه 3 تیر داریم یا نه. دقت کن هر ادراک یک جمله مثل منطق گزاره ای. یک آرایه به عنوان همه خونه هایی که امن هستند باید نگه داریم یک آرایه هم میشود واسه اون خونه هایی که هنوز ملاقات نکردیم. راجب خونه های امن هم عامل مرتب از پایگاه دانش میپرسد و استنتاج میکند. نکته: چند مدل داریم یعنی چند حالت وجود دارد که به ازای انتساب هایی که به متغیر ها میدهیم در آن همه کلاز ها ارضا پذیر هستند. نکته: 4 متغیر داشتیم 16 حالت برای مقدار دهی وجود دارد. کلا 16 مدل وجود دارد بهتر هست. دقت کن حتما لازم نیست از رزولوشن استفاده کنی کل راه بلدی. نکته: وقتی یک جمله دادند تو ببین نسبت به چه حالت هایی میتواند غلط باشد و اگر نتوانستی هیچ حالتی پیدا کنی که کل جمله غلط شود میشود تاتولوژی یا ولید یا معتبر و اگر پیدا کردی که جمله درست باشد میشود نا معتبر یا شاید هم قابل ارضا. ببین وقتی هیچ حالتی نتونیم پیدا کنیم که کل جمله رو غلط کنه میشود تاتولوژی. مثلا اونجا چون شرطی بود باید سمت راستی درست و سمت چپی غلط باشد. برای DPLL باید استنتاج رو تبدیل کنی به اینکه KB AND NOT ALPHA ارضا ناپذیر باشد یعنی خروجی باید شکست بخورد. دقت کن الگوریتم رزولوشن در بدترین حالت میتواند نمایی باشد رشدش و نیاز نیست همه جا استفاده کنیم اکثر پایگاه دانش ها هم فرم خوبی دارند. فرم خوب همان horn و عبارات معین هستند عبارات معین یعنی اینکه دقیقا یک دونه لیترال مثبت دارد. تک لیترال هم عبارت معین دارد. هورن یعنی میتواند لیترال مثبت نداشته باشد هر عبارت معین میتوانند هورن باشد. جمله ای که لیترال مثبت نداشته باشد کلاز گل معنی میشود. دقت کن به ازای دو تا فرم هورن حداکثر دو تا لیترال مثبت داریم حاصل اعمال رزولوشن هم روی دو عبارت هورن قطعا هورن میماند چرا چون عبارت حاصل قطعا یک لیترال مثبت کمتر دارد. عبارات معین را میتواند به شکل شرطی نوشت به طوری که سمت چپ and یک سری عبارت هستند و MP میزنیم خیلی راحت. دقت کن لیترال مثبت سمت راست عبارت implication میفتد. مرتبه این کار میشود خطی که از forward, backward chaining استفاده کنیم دقت کن با استفاده از این 2 الگوریتم عبارات هورن. فکت: تک لیترال مثبت سمت چپ ارضا پذیر و درست باشد سمت راست را به عنوان فکت جدید اضافه میکند. تو back به صورت بازگشتی میایم پایین تا به فکت ها برسیم و هر سری سمت راستی را مبدا قرار میدهیم که در سمت چپ هدف موجود باشد. این backward حتی کمتر از خطی هست چون اونایی که به هدف نیاز ندارند را نگاه نمیکند ولی forward همه را نگاه میکند. دقت کن همه این موارد داخل sat به اندازه 2n فضای حالت دارند. یکی از تکنیک های تسریع برای الگوریتم DPLL خاتمه سریع هست به این شرح که ما با ازای هر مقدار کلاز های باقی مانده را نگاه میکنیم اگر باعث غلط شدن کلازی شود دیگر کل اون زیر شاخه را نگاه نمیکنیم چون مشخص هست جوابی نداریم و به سراغ زیر شاخه مثلا فالس میرویم دوما چون هر کلاز از OR یک سری عبارت تشکیل شده است اگر درستی یک لیترال ثابت شد کافی هست و میگیم کل این کلاز ترو هست. در دومین هیورستیک دقت کن اگر به همه اون نماد خالص ها لیترال درست دادی و مقدار ترو گرفتن و جواب نداشت دیگه سراغ فالس دادن نرو چون میدانی اگر جوابی بود قطعا تو ترو بدست می آمد. و اول هم باید به همین نماد خالص ها مقدار بدی. کلاز واحد مثل نماد خالص فقط مقداری که آنها را ترو میکند در نظر میگیرد اول هم به اینها مقدار میدهیم. انتشار واحد: مقدار دهی به تک لیترال ها باعث تولید کلاز های واحد بیشتر میشود.

**فصل هشتم:**

زبان منطق گزاره ای برای نمایش محیط های پیچیده مناسب نیست و منطق مرتبه اول قدرت نمایش بهتری برای بازنمایی دانش دارد. و اینجا هم باید سینتکس و سمنتیک این زبان را یاد بگیریم. مشکلی که زبان های برنامه نویسی دارد این است که امکان استنتاج از دانش های قبلی ندارد و توابع وابسته به دامنه هست. در مرتبه گزاره ای همه چیز یا درست هست یا غلط ولی بعضی اوقات ما نیاز داریم به جزئیات بیشتر. مثلا نیاز داریم به دوستمون بگیم تلویزیون ببین نمیشه بریم داخل پایگاه دانش بگذاریم فقط ببین معنی نمیدهد پس همه چیزش خوب هست جز مشکل اینکه واسه محیط پیچیده کافی نیست. مثلا در منطق گزاره ای واسه ومپوس باید واسه هر 16 تا خانه قانون مینوشتیم ولی در زبان انسان فقط میتوانیم بگیم بو اومد ومپوس هست. پس ما نیاز داریم که زبان انسان و منطق گزاره ای را تبدیل کنیم یعنی اعلانی بودن و ترکیبی بودن و مستقل از متن را از منطق گزاره ای داشته باشد و قدرت بیان کافی را هم از زبان انسان داشته باشد. در زبان انسان یک سری توابع وجود دارد که برای هر ورودی یک خروجی دارد. مثلا پدر یک نفر فقط یک نفر هست. خروجی ها آن یکتا هست.

منطق گزاره ای فقط میگوید در جهان هستی فقط فکت هست که یا درست هست یا غلط ولی در منطق گزاره ای یک سری روابط و اشیا را هم پوشش میدهد و به صحیح غلط خلاصه نمیشود و بعضی چیز ها را نمیداند. در مدل منطق گزاره ای مرتبه اول، اول باید اشیا مدل خودمون را تعریف کنیم که دامنه ما میشوند. یک شی دقیقا فقط با یک شی دیگر رابطه دارد. مدل میاد هر کدوم از نماد هارا مثلا علی که برای انسان هست یا رابطه ها یا فانکشن ها را لینک میکند لینک کند به اشیا و رابطه ها در دنیایی که داشتیم به هر لینک کردن میگوییم یک تفسیر. مثلا یک تفسیر این هست که پی ترو هست یا بی فالس هست. پس مدل میشود جمع اون روابط و اشیا و فانکشن ها و این تفسیر ها که مشخص میکند هر اشیا با چه نماد هایی بهم وصل شده اند اینجا هم باید تحت یک تفسیر بگوییم مثلا یک رابطه درست یا غلط هست یا یک جمله مثلا اینکه علی و حسن با هم برادر هستند اگر علی واقعا به یک شخص و حسن واقعا به یک شخص اشاره کند تحت این تفسیر درست هست. پس نگاه کن اول اشیا و رابطه ها را تعریف کن بعد بیا یک تفسیر ارائه بده که ببینیم اون تفسیر نسبت به اون چیزهایی که تعریف کردیم درست هست یا نه. مثلا نگاه کن اگر یک تفسیری باشد که بگوید حسن روسری دارد این درست نیست یعنی وقتی لینک میکنیم درست نیست روسری داشتن باید به فاطمه لینک شود. تفسیر منظورم لینک ها هست بعد دقت کن مدل ها در منطق گزاره ای مرتبه اول بینهایت هست چون از قبل که اشیا مشخص نیستند بعد دقت کن اون منطق گزاره ای بود که دنیا ما 2^n حالت داشت چون فقط صحیح غلط داشتیم اینجا به اینگونه نیست. این سمنتیک بود. حالا سینتکس: term چیزی هست یک عبارتی هست که به یک ابجکت اشاره میکند مثلا برادری علی و حسین ممکن هست به دو اشیا در دنیا اشاره کنند. رابطه بین دو اشیا هست ولی تابع واسه یک اشیا و فقط یک خروجی دارد. مثلا رابطه برادری به هر تفسیری ممکن هست وصل شود ولی تابع فقط به یک نفر. جملات اتمیک عین منطق گزاره ای که پی و نات پی بود نیستند اینجا اون پریدیکت ها هستند مثلا یک رابطه یا یک تابع کلا اون داخل ترم ها هستند که ترکیبی از نماد ثابت یا تابع باشند. درستی غلطی هم گفتیم دیگه اگه طبق اون تفسیر اون رابطه برقرار باشد ترو هست اگر نباشد فالس. جملات پیچیده هم با عملگر های منطقی میان اضافه میشوند. معنی هم مشخص هست دیگر مثلا اور وقتی درست هست که یکی درست باشد. در اینجا برخلاف منطق گزاره ای از سور ها میتوانیم استفاده کنیم مثلا قبلی نیست که اگر لازم باشد چیزی را به همه بگوییم نتوانیم یا تک به تک نام ببریم. سور عمومی: برای همه اشیا دنیا گفته میشود. دقت کن که پی برای هر شی در دامنه ما درست هست. مثل اشیا چی بودند؟ علی شیردل پادشاه حسن پای چپ علی پای چپ حسن و تاج. اینها اشیا بودند. و مثلا توی اون سور هر شی را جای x قرار میدهی و اگر به ازای هر 5 تا درست بود همه درست میشوند. شرطی: چپ غلط باشد کلا درست هست. پس درستی سور عمومی به همین شکل هست دونه به دونه اشیا را جای x میگذاریم ببینیم درست هست یا نه. برای هر ایکسی که پادشاه هست میفهمیم که ایکس شخص هم هست. این فرق میکند با اند ها. اینجا یعنی هم پادشاه باشد هر ایکس هم شخص. سور وجودی: راجب برخی هست اون حقیقت یعنی یکی درست باشد کافی هست. استفاده شرطی با سور وجودی بیش از حد ادعا ضعیف میکند مثلا از تاج بودن میتوانیم انسان بودن را نتیجه بگیریم که مسخره هست واسه همین هست که سور وجودی همیشه با اند میاد بخاطر این علت هست. سور های تودرتو: یکی باشند تفاوتی ندارد ولی متفاوت باشند فرق میکند. دقت کن هر متغیر به داخلی ترین سور یا نزدیک سور به خودش اشاره میکند. بهتر است یک اسم دیگر استفاده کنیم. سور ها با استفاده از نقیض به یکدیگر میتوانند تبدیل شوند. برای تبدیل شدن بهم دیگر هر المان را نات کن.

دو شی وقتی با هم به یک چیز اشاره کنند با هم مساوی هستند. دقت کن لازم نیست حتما اون شرط آخر را بنویسی که تاکید کنی میتوانی شرط بگذاری که اگر از نماد های خاص استفاده کردی پس دقیقا هر نماد به یک نفر اشاره میکند شرط دوم closed world هست میگوییم وقتی به 2 برادر اشاره کردی پس هیچکس دیگر برادر این یارو نیست. یعنی فرض میکنیم دامنه کلا همون 5 تا شی هستند. بهش database semantic میگوییم این ویژه تر هست نسبت مثلا دو اسم خاص میتواند به یک نفر اشاره کند در زبان معمولی. پریدیکت ها به صورت نامحدود میتوانند argument بگیرند. اگر سور نداشتی برو ببین اون رابطه تفسیری که دارد درست هست یا نه. پریدیکت میتواند ورودی نداشته باشد و مثل منطق گزاره ای همون ترو فالس باشد. پایگاه دانش عین همون منطق گزاره ای هست. با TELL اضافه میکنیم و با ASK میپرسیم. استنتاج هم عین همون روش های قبلی هستند. پریدیکت ها یک سری یکتایی میتواند باشد دوتایی هم داریم دقت کن تابع فقط یک خروجی دارد و یکتا هست. شرطی از مقدم تالی را نتیجه میگیری. با سور عمومی اند نمیاد. ببین جمله که دادن دیگه به سور نات فکر نکن به همون سور فکر کن در همون شرایط مثلا اگر نات سور وجودی باید شرطی اومده بود خوب مشکل دارد. شرطی: درست بودن مقدم تالی را نتیجه میگیریم. وقتی نماد خالی بیاد چون ربطی به متغیر نداریم و مستقل از متغیر هست پس تالی رو پرانتز بندی کن. وقتی تابع میاد با متغیر یعنی همه یا هر ولی وقتی نمیاد کلا یک تک نماد هست که خوب معنی متفاوتی دارد. یعنی اون سور عمومی به همون مقدم ربط دارد و دیگر ربطی به تالی ندارد. ما اضافه کردن متغیر در منطق گزاره ای نداشتیم. ولی قدرت منطق مرتبه اول این هست که میتواند متغیر داشته باشد. پس اگر جملات متغیر نداشتند میتوانیم به منطق گزاره ای تبدیل کنیم. برای بررسی کردن تاتولوژی یا ولید بودن بیا از لحاظ معنایی تفسیر کن.

**فصل نهم:**

بخش اول: حالا سراغ استنتاج در منطق مرتبه اول میرویم. و جملاتی که سور دارند و متغیر ندارند را به منطق گزاره ای تبدیل میکنیم و وقتی تبدیل شد با همون روش های استنتاج حل میکنیم ولی این کار خیلی کارآمد نیست و زمان بر هست بخاطر همین قوانینی میگوییم که روی همون مرتبه اول کار میکند و یک سری الگوریتم هایی که روی فرم horn کار میکند میرسیم بعد به الگوریتم کامل رزولوشن در منطق مرتبه اول میرسیم.

ترم گراند، ترمی است که متغیر ندارد. ترم چی بود؟ نماد های ثابت یا نماد های تابعی یعنی جای متغیر ها همون اسم های خودمون را قرار بده. و اینگونه جملاتی تبدیل میشود که متغیر ندارند و یک سری گزاره هست به شکل نماد گزاره ای و اینجوری کل پایگاه دانشی که یک سری گزاره ای دارد و از قوانین استنتاج میتوانیم استفاده کنیم.

حذف سور های وجودی: ساده هست، سور را حذف کن و جای اون متغیر بیا یک اسمی بگذار که قبلا بکار نرفته باشد. به این نمادی که اتلاق میکنیم نماد اسکالم هست. دقت کن جمله جدید از نظر منطقی هم ارز با قبلی نیست ببین دقت کن تو قبلی سور عمومی از نظر منطقی هم با هم، هم ارز بودن فقط این رو بدون که اگر جمله اول ارضا پذیر باشد جمله حاصل شده با ثابت اسکالم نیز قابل ارضا شدن هست.

برای ارضا پذیری یک جمله یک مدل ارائه بدهیم مدل همان تفسیر ها (جملات) و لینک ها بودند رابطه هم که یادت هست یک سری کلا تاپل بود. ببین p میتواند به هر ابجکتی اشاره کند ولی منتها وقتی g درست هست که c1 دقیقا به همون ابجکتی که میخواهد اشاره کند این ارضا پذیر هست، ارضا پذیر با درستی یکی نیست، ارضا پذیر یعنی این جمله یک مدلی دارد که تحت اون درست است. اگر یک جمله مدلی داشته باشد که بتواند نسبت به اون ارضا پذیر باشد اونوقت وقتی اون سور وجودی را با یک نماد جا به جا کنیم جمله حاصل هم ارضا پذیر هست. هر پایگاه دانش مرتبه اول را به فرم منطق گزاره ای به طوری که استلزام را حفظ کرد تبدیل کرد یعنی اگر با یک جمله اونجا میشد استلزام کرد اینجا هم میشود. مشکل ما با نماد های تابعی هست، قبلی ها را با ترم گراند جا به جا میکردیم مشکل این هست که ما به تعداد نا محدودی میتوانیم ترم گراند را روی این تابع تکرار کنیم و پایگاه دانش منطق مرتبه اول نا محدود میشود و این باعث میشود استلزام سخت شود. نکته این هست که اگر بتوانیم نسبت به الفا استنتاج کنیم به قسمت محدودی هم استنتاج میشود. یعنی اول در عمق صفر میرویم و با نماد های تابعی کاری نداریم میبینیم اون جمله بدون اینا قابل استنتاج هست یا نیست اگر نبود یک عمق میایم بالاتر و یکبار اون تابع را اجرا میکنیم و نسبت به اون استنتاج میکنیم اگر بدست اومد که هیچی اگر نیومد یکی عمق را افزایش بده همینطوری اگر نشد برو تا بینهایت پس اگر الفا ما واقعا قابل استنتاج باشد تا تعداد محدودی جواب میدهد ولی اگر واقعا استنتاج نشود تا بینهایت میرود برعکس رزولوشن در منطق گزاره ای که به ما میگفت جمله استنتاج نمیشود با no. بخاطر همین استنتاج در منطق مرتبه اول نیمه تصمیم پذیر هست. تصمیم پذیر یعنی اینکه برای یک مسئله ما الگوریتمی داشته باشیم که اگر جواب داشت yes و اگر جواب نداشت نه برگرداند. مسئله استلزام دقیقا همچین هست یعنی هیچ الگوریتمی وجود ندارد که به ما جواب بله یا نه بدهد بخاطر نماد های تابعی هست چون بینهایت بار اجرا و گزاره های جدید میدهند واسه همین الگوریتم های رزولوشن کامل کاذب هستند. مشکلات دیگری دارد این هست که جملات غیر مرتبطی زیادی تولید میکند این روش گزاره سازی. یعنی مثلا ما واسه علی میخواهیم ولی واسه حسن هم میسازد در صورتی که واسه حسن نیازی نداشتیم. تعداد هم p.n^k، p تعداد پریدیکت ها هستند n تعداد نماد های ثابت و k تایی هست چون هر کدوم k تا حالت دارند دیگه. واسه همین دنبال روش های مستقیم در منطق مرتبه اول داریم.

قوانین استنتاج: در GMP که همان modes ponens خودمان هست بیا یک انتسابی به اون متغیر ها بکن که به ازای آن قسمت تالی شرط درست در بیاد. در واقع به ازای اون جانشینی ها نسبت به متغیر ها با نماد های ثابت مقدم درست در بیاد و به واسطه اون تالی هم درست در بیاد. Substitution تتا روی چیزی یعنی اون جانشینی ها را جای مثلا اون چیزی که داری بگذار و ازش نتیجه بگیری. یعنی مثلا هر جا ایکس دیدی جاش بگذار حسن هر جا وای دیدی بزار علی. به این کار یکسان سازی یا unification گفته میشود کلا این تابع قرار است دو تا ورودی بگیرد و یک جانشینی برگرداند که 2 تا عین هم شود. دقت کن مثلا وقتی y میگذاریم در یکسان سازی منظورمان سور عمومی هست پس وقتی همه علی را میشناسند پس حسن هم علی را میشناسد.

دقت کن در دو جمله در یکسان سازی نباید از متغیر های مشترک استفاده کنی، و استاندارد سازی به حذف اشتراک اسامی متغیر ها گفته میشود پس دقت کن نیاز نیست متغیر ها اسامی یکسان باشند. 3 قانون برای استاندارد سازی داریم: 1. اگر دو پریدیکیت یا دو تابع دادند حتما باید یکسان باشد یعنی هر دو مثلا f باشند و گرنه خطا. 2. دو ترم ثابت قابل یکسان سازی نیستند مثلا علی با حسن. 3. هنگام یکسان سازی یک متغیر و یک ترم باید چک کنیم آن متغیر در ترم ظاهر نشده باشد یعنی اینکه چون هعی باید این 2 تا را بهم یکسان کنی در لوپ بینهایت گیر میکنی. دقت کن وقتی میگویی جای z بزار a یعنی دیگه هر جا z دیدی تعویض کن.

ترتیب جانشینی فرقی نمیکند از کدوم طرف شروع کنی. نماد های تابعی باید یکسان باشند دقت کن. دقت کن که یک نماد ثابت را با یک نماد تابعی نمیتوانی یکسان کنی چرا؟ چون مثلا نمیتوانی بگویی مادر رضا، زهرا هست چون ممکن است رضا به کسی دیگر هم در اون دنیا اطلاق شود. کلا دنبال جانشینی هستیم تا بتوانیم تالی را نتیجه بگیریم و اساس منطق مرتبه اول هست. یعنی وقتی درست باشد اون جانشینی اون SUBST استلزام میشود از اون شرطی ما. یکسان سازی شکل های متفاوتی میتواند داشته باشد یعنی جانشینی ها مختلف بدهیم که یکی از آن یکسان سازی ها کلی تر هست و هر دو عبارتی یک یکسان ساز کلی دارد. از اون یکسان ساز کلی بقیه رو میتوان نتیجه گرفت.

عبارت معین: ترکیب فصلی گزاره ها بودند که یکیش به صورت نقیض می آمد که میشد به صورت شرطی نوشت. کلاز های تک لیترال و مثبت را فکت میگفتیم. دقت کن تو جملات شرطی اینجا میتوانیم از متغیر هم استفاده کنیم. مثل منطق گزاره ای همه پایگاه دانش نمیتواند به صورت این شکلی باشد و خوبی همین به شکل شرطی نوشتن هست. دقت کن نماد تابعی نداریم و همه عبارت معین هستند. FC: در هر دور میاد چک میکند که سمت چپ برقرار هست اگر بود سمت راست هم برقرار هست به یک شکل همون GMP هست. پس باید یک جانشینی واسه سمت چپ پیدا کنیم تا باهاش راست را نتیجه بگیریم. بعد این جملات جدید که توانستیم نتیجه بگیریم را اضافه میکنیم و بعد با اطلاعات جدید باید ببینیم میتوانیم یک جانشینی پیدا کنیم که سمت چپ درست باشد و سمت راست درست شود. Data log میگوید که اگر جملات پایگاه دانش ما عبارت معین باشند و هیچ تابعی و نماد تابعی نداشته باشد، خوبی این هست که الگوریتم FC براش کامل هست و صحیح هست یعنی اگر بتواند استنتاج کند بعد از یک مدتی بله میگوید و اگر نتواند نه میگوید اما اگر تابع داشته باشد حتی اگر همه عبارت معین باشند الگوریتم ممکن است جواب نه را ندهد و تمام نشود. هیچ الگوریتمی نداریم که بتوان جواب نه از استنتاج را به ما برگرداند حتی اگر پایگاه دانش ما عبارت معین باشند. بعد از چند دور این الگوریتم تمام میشود P.N^K، برای بهبود کارایی FC: ما کلا میخواهیم یک سری تطابق الگو بدهیم بین سمت چپ عبارات شرطی و جملات درون پایگاه دانش، یعنی مثلا اگر ما 10 تا موشک داشتیم 10 بار انجام میشود و نتیجه میگیریم هر 10 تا موشک اسلحه هم هستند، ولی اگر عبارات سمت چپ ما متغیر های زیادی داشتند از اونی شروع کن که کمترین عبارت و متغیر را دارد مثلا 5 تا موشک دارد حالا چک میکنیم این 5 تا موشک واسه روسیه هست یا نه جا اینکه بیایم 100 تا چیز که واسه روسیه هست رو بررسی کنیم که ببینیم واقعا هست بعد بررسی کنیم که موشک هم هست. پس دنبال بهترین ترکیب هستیم و این مسئله از مرتبه NP-HARD هستیم پس باید دنبال هیورستیک باشیم مثل فصل 6 دنبال اونی هستیم که متغیر کمتری دارد. هر فکت جدید پیش زمینه ای دارد در فکت هایی که دور قبلی بدست آمده است پس بنابراین برای اجرای دور دوم فکت های اولیه را در نظر نگیر چون فکت های تکراری میدهد و خوب اون جدید هم اگر قابل بدست آوردن بود مشخص بود پس فقط سمت چپ را با فکت های جدید در نظر میگیریم. الگوریتم FC مانند proposition logic یک سری جملات بدرد نخور هم تولید میکند. یک ایده که شبیه به BC هست این هست که تو مثلا دنبال این هستی که روسیه دشمن هست؟ خوب بیا تمام جملات را در نظر بگیر که سمت راست آن به دشمن بودن روسیه اشاره کرده باشد، و بیا یک پریدیکیت magic x اضافه کن. و دقت کن این دقیقا باید برای روسیه باشد یعنی فقط به ازای x مساوی روسیه نتیجه میگیریم و ارضا میکند پس فقط اون رو در نظر بگیر ببین ترو میشود یا نه. در واقع اون جملاتی که انتخاب کردیم رو یک جادو براش در نظر میگیریم. BC: یک هدف داریم بعد میبینیم این هدف سمت راست کدوم جملات هست بعد بیا انتخاب کن بعد برو سراغ اثبات سمت چپ پس به شکل AND-OR هست. بعد دقت کن سمت راست را برداشتی باید یکسان سازی و جانشینی کنی دیگه بعد جملاتی که بدست میاد از چپ را به پایگاه دانش اضافه کن. فقط حواست باشه وقتی جانشینی میکنی تو جملات جدید که نتیجه میگیری لحاظ کنی. هعی سمت راست بگذار جانشینی کن بعد برو سراغ سمت چپ. این الگوریتم چون عمقی هست پس خطی هست. کامل نیست چون ممکن هست توی دور بیفتد و ربطی به تابع ندارد. یک مشکل دیگر محاسبات تکراری هست پس بهتر هست کش کنیم. PROLOG: یک زبان برنامه نویسی منطقی هست مبتنی بر BC، اینجا ثابت ها را کوچک و متغیر ها با حروف بزرگ نوشته میشود. همچنین همه به شکل عبارات معین یا شرطی هستند هم عبارات برعکس نوشته میشود. ( این قسمت تست هاش مجددا بررسی شود.)، قانون رزولوشن همون قبلی هست، منتها چیزایی را حذف میکنیم که علامت ها نات هم باشند و همچنین قابل یکسان سازی باشند، دقت کن جانشینی داریم و باید استاندارد سازی کنیم یعنی باید اسم متغیر ها را عوض کنیم تا به مشکل نخوریم، فکتورینگ: تکراری ها را نمینویسیم باید انجام بدهیم تا خاصیت کامل بودن حفظ شود. اینجا میگوید دو تا عبارتی که میتوانی یکسانی کنی با هم را فقط یکبار بنویس. چون برای رزولوشن نیاز هست جملات CNF باشند پس ما اینجا میتوانیم همه جملات را تبدیل کنیم به CNF. فقط باید دقت کنی هر متغیری میبینی اینها قبل آن سور عمومی بوده اند نه با سور وجودی، هر وقت سور وجودی داخل سور عمومی داشتیم میایم یک تابع اسکالم برحسب متغیر سور عمومی میگذاریم. پس برای رزولوشن اول اون جملاتی که داشتیم را به CNF با همون قانون ها تبدیل میکنیم بعد نات الفایی که میخواهیم اثبات کنیم را به پایگاه دانش اضافه میکنیم بعد میایم رزولوشن میزنیم. وقتی به تهی رسیدیم ببینیم اون متغیر ما توسط چه کسی جانشین شده است هر کی شد اون مد نظر ماست.

فصل نهم بخش دوم:

کامل بودن رزولوشن به این معنا هست که اون kb و نات الفا را فالس میدهد قطعا اگر قابل استنتاج باشد. کلا مسئله استنتاج یک مسئله نیمه تصمیم پذیر هست به خاطر همون تابع ها. رزولوشن کامل هست به این معنی که اگر الفا را بدهیم جواب را میدهد نه به این معنی که ما هعی جملات جدید تولید کنیم. Unit preference میگوید اول دنبال تک لیترال ها برو و رزولوشن بزن تا بهی تهی برسی البته همیشه کامل نیست چون ممکن هست اصلا به شکل تک لیترال نباشد البته به شکل horn باشد امکان پذیر هست به صورت تک لیترالی. دقت کن مجموعه دقیقا باید جوری باشد که بتوانی استنتاج کنی. این قضیه توی DPLL هم بود که شبیه forward chaining بود. رزولوشن ورودی: عباراتی که انتخاب میکنی حتما باید داخل پایگاه دانش هم باشد فقط هم در حالت horn کامل هست و این روش هم کامل هست.

**فصل دهم بخش اول:**

تا الان 2 تا عامل داشتیم یکی عامل حل مسئله که میومد جستجو میکرد و به یک دنباله اکشنی میرسید به این دنباله ما میگوییم پلن، مشکل اصلی این بود که از حالت های مسئله چیزی نمیدانست و مجبور شدیم یک سری هیورستیک بدهیم به عامل که سرچ کارا تر شود مشکل این بود که هیورستیک فقط مختص به همون مسئله بود و تو مسئله های دیگر نمیشد استفاده کرد و وابسته به دامنه همان مسئله بودند. مشکل عامل مبتنی بر منطق این بود که جملات ما نمی توانستند متغیر داشته باشند و هر اکشن برای هر خونه دقیقا مشخص میشد و اندازه n^2 باید جمله می نوشتیم.

حالا اینجا میخواهیم با یک توصیف ثابت یک اکشن را مشخص کنیم. PDDL: ترتیب عطفی از جملاتی بود که گراند بودند یعنی متغیر نباید داشته باشند و متغیر هم نباید داشته باشیم تابع هم نباید داشته باشیم. همچنین از data base semantic استفاده میکنیم و هر لیترالی که بهش اشاره نشده است مقدارش فالس هست و نمینویسیم و فقط چیز های مثبت مینویسیم و نقیض نمیاد. دو اسم مختلف هم به دو شی مختلف اشاره میکنند. و مزیت این هست که هر اکشن را با جزئیات میتوانیم نشان دهیم یک شما کلی میشود.

یک سری پیش شرط داریم مثلا اگر پرواز از شهر الف به ب برود شهر الف و ب باید فرودگاه داشته باشد و ما هواپیما هم باید داشته باشیم همچنین اثر این اکشن هم هست که اگر پرواز کردیم هواپیما در شهر ب هست و در شهر الف نیست. و جملات متغیر دار را گزاره ای میکنیم. یک سری جانشینی هم داریم که دیگه جای اون متغیر ها قرار میگیرند و تبدیل به جملات گراند میشوند.

نمایش factor این بود که میایم و از داخل حالات مسئله یک سری اطلاعات بیشتر بدست میاوردیم و اونطوری حل میکردیم و خیلی بهتر از عامل های جستجو میشد. یک زبان توصیف میکنیم تا برای حالات مسئله یک سری اطلاعات بیشتر بدهیم و قدرتی اندازه منطق مرتبه اول دارد تا بتوانیم از هیورستیک ها بهتر استفاده کنیم. فلوئنت همان لیترال هایی هستند که در جهان تعریف کردیم و مقدارشان میتواند تغییر کند بدون متغیر هست یعنی حروف کوچک نداریم مثل x y و فاقد تابع هستیم و به هر لیترالی که نیامده است یعنی برقرار نیست و فالس هست. خود اسم مختلف نشان دهنده یکی نبودن هست برخلاف منطق مرتبه اول که باید حتما اشاره میکردیم. نات یک لیترال هم نمیاوریم چون اصلا وقتی میخواهیم بگیم ارزش فالس دارد اصلا نمیاوریم. در تعریف اکشن ها میتوانیم از متغیر استفاده کنیم برخلاف گزاره ای.

اکشن الف قابل انجام است اگر پیش شرط های آن ارضا شده باشد مثلا تو مثال اول چطوری فهمیدیم؟ چون توی عبارتی که به ما دادند نیامده است طبق قاعده database semantic چون نیامده است پس میتوانیم این اکشن را انجام دهیم. حالا تابع ریزالت: دقت کن تابع effect روی پیش شرط ها تاثیر میگذارد با این ها کار میکنیم و تابع ریزالت را میسازیم که به 2 بخش ADD که یک چیزی به خروجی اضافه و DEL یک چیزی از خروجی کم میکنند هستند. اکشن چطوری انجام میشود؟ هر لیترال مثبت باید در اون s که پیش شرط باید باشد و هر لیترال منفی نباید در پیش شرط های ما باشد البته مشخص هم هست زیرا که اصلا منفی نمیاد که بخواهیم باشد دقت کن اگر پیش شرط ما نقیض یک لیترال باشد و اون در حالت s به صورت مثبت آمده باشد یعنی اکشن قابل انجام نیست. دقت کن نتیجه وقتی هست که ما یک دنباله ای از اکشن ها داشته باشیم که هدف را ارضا کند و دقت کن قبل هدف سور وجودی میاد میگیم هر حالتی که این را استلزام کرد بپذیر. هر حالت هم گفتیم and یک سری عبارت فلوئنت بدون تابع به شکل مثبت هست. دنبال این پلن هستیم که با کمترین هزینه هدف را ارضا کند.

در منطق گزاره ای این مشکل داشتیم واسه چیز هایی که تغییر میکردند در هر زمان یک جمله مینوشتیم اینجا اون مشکل زمان را نداریم و پیش شرط این هست که ما الان در حالت T هستیم و وقتی انجام میشود در حالت T+1 هستیم این به صورت ضمنی هست. هدف میتواند متغیر و لیترال منفی هم داشته باشد. خروجی این هست که هدف را استلزام کند که درست هست. اگر نماد های تابعی را اضافه کنیم مسئله شاید به جواب نرسد و تو لوپ گیر میکنیم و plan SAT سخت تر میشود. هر دو مسئله PLANSAT و BOUNDED PLANSAT تصمیم پذیر هستند یعنی یک مسئله بدهیم و مشخص میکند که با یک پلن میتوان حل کرد یا نه. مسائل نیمه تصمیم پذیر مثل استنتاج در منطق مرتبه اول الگوریتمی داشتیم که جواب بله را تایید کنند ولی تو تمام نسخه ها نمیتواند جواب نه را تایید کند، تو مسائل تصمیم ناپذیر حتی جواب بله را هم نمیگویند.

هیچکدوم از الگوریتم ها در زمان کمتر از نمایی تضمین نمیکنند چون NP-hard هستند. البته برای خیلی از مسائل نمونه PLANSAT میتوانیم پلن پیدا کنیم تو کمتر از نمایی و خوب هست. اکشن را چه زمانی میتوانیم انجام دهیم؟ وقتی پیش شرط های مثبت در اون اکشن آمده باشند و پیش شرط های منفی نیامده باشند. خیلی وقت ها اکشن هایی را اعمال میکنیم که اصلا نیازی نداریم بهشون. مشکلی که دارد این هست که در فضای حالت بسیار بزرگ کارا نیست. و اکشن های نا مربوط را بدون داشتن هیورستیک خوب انجام میدهد. روش backward: از هدف شروع میکنیم و میگوییم چه حالت هایی قابل انجام بودند که من را به حالت هدف فعلی رسانده اند به این اکشن ها relevant گفته میشود. دو شرط این هست که 1 حالت هدف اشتراک با add list تهی نباشد و با لیست حذف اشتراک نداشته باشد. در واقع باید ببینیم add list کدام یک با هدف ما قابل یکسان سازی هست. در back ward از هیورستیک ها نمیتوانیم استفاده کنیم بخاطر اینکه با یک مجموعه تمام حالات کار میکند. هزینه مسیر یا هزینه راه حل همان تعداد اکشن ها هستند در هیورستیک های ما در اینجا و همیشه این هیورستیک قابل قبول هست. مسئله این هست که بیا پیش شرط ها را نادیده بگیر بعد ببین با این اکشن ها چند حرکت نیاز هست تا حل شود. البته باز هم این مسئله NP-HARD هست چون ما نمیدانیم کدام اکشن ها را انتخاب کنیم تا به هدف برسیم و نیاز به یک هیورستیک قابل قبول داریم. یک هیورستیک دیگر داریم برای این هیورستیک باید بتوانیم که مسئله را به زیر اهداف تقسیم کنیم و هر کدام از اینها را بدست بیاوریم یعنی زیر اهداف ما کاملا مستقل از همدیگر تعریف شده باشند. البته شاید این هم قابل قبول نباشد چون با یک حالت مثلا برویم و هزینه کمتری داشته باشد. یا مثلا هیورستیک ما بشود بشینه هزینه رسیدن به یکی از زیر اهداف. دقت شود هر کدام از این حالات یک مجموعه حالت هستند چون یکبار مثلا میتواند هواپیما الف باشد یکبار هواپیما ب. هیورستیک اینجا زیاد معنی نمیدهد چون ما یک مجموعه حالت داریم نه یک حالت که هیورستیک بدهیم پس forward بهتر هست. max چند تا هیورستیک قابل قبول حتما هیورستیک قابل قبول هست.

فصل دهم بخش دوم:

گراف برنامه ریزی: اینجا چندین عامل داریم که دنبال هدف میگردند و هزینه آن کمتر هست چون سریعتر میشود تا نسبت به اینکه فقط یک عامل شروع کند به سرچ کردن. پیش شرط ها وارد میشوند و از طرف دیگر effect ها خارج میشوند و به عنوان پیش شرط های بعدی در نظر گرفته میشوند همچنین با اکشن no-OP میگوییم هر چی الان برقرار هست در زمان بعدی هم برقرار هست یعنی اگر در P1 برقرار هست در P2 هم برقرار هست. effect ها میتوانند نات هم باشند صرفا میخواهیم بگیم با چه اکشنی چه نتایجی رخ میدهد و در گراف قرار میدهیم. همچنین در یک حالتی یک لیترال ممکن است ظاهر شود ولی در واقع تو مسئله ما نتواند در اون لحظه درست باشد و برقرار باشد یا در همان لحظه انجام نشود و در لحظات بعدی برقرار باشد چرا چون ما چندین عامل داریم که دارند کار میکنند جا یک عامل و دقت کن هیچ موقع دیر تر از زمان واقعی نمیشود دقیقا به همین علت چند عاملی بودن. اگر در گراف در هیچ لولی یکی از این g ها ظاهر نشود میتوانیم به قطعیت بگوییم که در مسئله واقعی هم ما نمیتوانیم به آن g برسیم و میتواند غیر قابل حل بودن مسائل را نشان بدهد. ممکن هست یک لیترال داشته باشیم در گراف پلن ولی هیچ راه حلی برای رسیدن به آن نداشته باشیم اما برعکس این وجود ندارد یعنی اگر سوال حل بشود در گراف پلن به آن میرسیم، اگر 2 لیترال باشند که نتوانند همزمان درست باشند یا 2 تا اکشن باشند که یک پلن درست نتواند شامل آن ها شود باید mutex بگذاریم انگار این 2 طرف با هم مشکل دارند. یعنی نمیتواند مثلا هم کیک داشته باشیم هم کیک بخوریم. اگر پیش شرط ها mutex باشند اکشن ها هم mutex هستند. اگر دو اکشن تاثیر متناقض داشته باشند هم با هم mutex هستند. پلن ها باید به شکل سری باشد و هیچ تاثیری روی هم نداشته باشند چون در محیط واقعی مسئله فقط یک عامل داریم واسه همین mutex رخ میدهد. فکت یک اکشن نقیض پیش شرط یک اکشن دیگه هم باشد mutex هست. دو لیترال mutex هستند اگر هر جفت action یی که اینها را بدست میاورد mutex باشند.

Level off: تو گراف برنامه ریزی به دو سطح مشابه رسیدیم دیگر لازم نیست ادامه دهیم. مرتبه زمانی و حافظه از مرتبه چند جمله ای هست واسه همین میتوانیم هیورستیک استخراج کنیم و گراف را سریع بسازیم. از یک گراف برنامه ریزی میتوانیم تمام اطلاعات را استخراج کنیم. هزینه لول هم این هست که در گراف برنامه ریزی کجا ظاهر شده اند در کدام سطح و یک هیورستیک این هست که max این 2 را بگیریم که هیورستیک قابل قبول هست چون میدانیم همیشه زودتر رخ میدهد به خاطر چند عامل.

یک هیورستیک بهتر به این شکل هست که بین هر 2 اکشن mutex بگذاریم تا همزمان اجرا نشوند و اینقدر ادامه بدهیم که g ها بدون اینکه mutex یی ظاهر بشوند یعنی اولین لایه ای تمام زیر اهداف ها بدون mutex ظاهر شده باشند به عنوان جواب برگردانیم، بین no-OP ها mutex قرار نمیدهیم، اما این هیورستیک درست هست که خیلی خوب هست ولی کاملا خوب نیست چون مثلا 3 تا اکشن داریم که 2 به 2 با هم مشکلی ندارند ولی 3 تا با هم به مشکل میخورند چون پلن گراف 2 به 2 بررسی میکند و نمیفهمد. میتوانیم اینها را پیشگیری کنیم ولی نمیصرفه چون هزینه ساخت خیلی بالا میرود و اون توان 2 میشود توان 3 و 4 و .... یک راه دیگه داریم اینکه به جای اینکه توی اون گراف برنامه ریزی بگردیم بیایم توی یک گراف برنامه ریزی جدیدی که اندازش چند جمله ای هست و کوچکتر هست دنبال جواب بگردیم، یعنی بگرد دنبال اولین لولی که همه لیترال ها بدون mutex ظاهر شده باشند اگر به اون لولی نرسیده باشیم که g1 تا g3 باشند میفهمیم مسئله جواب ندارد. بعد که به اون لول رسیدیم استخراج جواب را شروع میکنیم و یک جستجو را انجام میدهیم که به عنوان جواب درست برگردانیم اگر رسیدیم که هیچ اگر نرسیدیم یک سطح دیگر اضافه میکنیم یا الگوریتم را تمام میکنیم. دقت کن یک زیر مجموعه از اینها را انتخاب میکنیم. Mutex نباشند میتوانند داخل یک پلن باشند. فاز جستجو ما نمایی هست. تا کی باید این الگوریتم را ادامه دهیم؟ مثلا level off رخ دهد یعنی هیچ چیز جدیدی نتوانیم اضافه کنیم. شرط واقعی و درست برای اتمام: وقتی نتوانیم به جواب برسیم یعنی به یک زیر مجموعه ای از زیر اهداف رسیدیم که نتوانستیم اکشن هایی را پیدا کنیم که mutex نباشند تا بتوانند زیر اهداف را ارضا کنند. دقت کن وقتی زیر اهداف ارضا نمیشوند برای سطح بعدی هم چک میشود و اگر نتوانیم در آن لول هم این زیر اهداف را نتیجه بگیریم یعنی کلا نمیتوانیم ارضا کنیم. وقتی الگوریتم خاتمه میابد که یا level off کنه و دیگر لیترال های ارضا نشده هم تغییری نکنند. لیترال ها به طور متناوب افزایش پیدا میکنند و هیچ چیزی حذف نمیشود و همچنین در لول های بعدی ممکن هست اون لیترال هایی که ارضا نمیشدند یا اکشنی قابل انجام نبود قابل انجام بشود. تعداد mutex ها هم بین لیترال ها هم بین اکشن ها کم میشوند یا ثابت میمانند دقیقا برعکس اکشن ها و لیترال ها. به زیر اهدافی که هنوز ارضا نشده اند no good میگوییم.

تبدیل مسائل برنامه ریزی به SAT: این روش حتی از گراف پلن هم بهتر هست چون الگوریتم های خوبی و سریع به جواب برسند مثل DPLL و این روش هست که برنامه ریزی را تبدیل کنیم به یک مسئله SAT بعد الگوریتم هایی که استفاده میشوند را اجرا کنیم اگر جوابی وجود داشت تبدیل میکنیم به برنامه ریزی البته درست هست که تو زمان چند جمله ای را تضمین نمیکنند ولی سریع به جواب میرسند. اول حالت اولیه را که یک سری لیترال هستند به شکل یک سری گزاره منطقی مینویسیم واسه هر لول از اکشن هم یک سری گزاره در نظر میگیریم و اگر در مسئله SAT مقدار این گزاره 1 شد باید در پلن ما باشد. باید از خود پلن به effect ها و پیش شرط ها یک لینک بکشیم و نشان بدهیم. دقت کن یک اندیس k هم باید در نظر بگیریم که بگیم تا این زمان باید در هدف باشیم. دقت کن اگر یک اکشن در زمان t+1 درست هست یا اون توی زمان قبلی هم درست بوده و هیچ اکشنی نتوانسته آنرا رد کند یا اکشنی باعث درست شدن این در زمان قبلی شده است.

و اگر یک اکشنی در زمان t ترو هست به این معنی هست که پیش شرط ها هم در همین زمان ترو بودند دقت کن در یک زمان نباید بگذاریم بیش از 1 اکشن اجرا شوند. دقت کن اون چیز هایی هم که برقرار نیست عین گراف پلن به شکل نات باید بیاریم. پلن هم میشود اکشن هایی که ترو شده اند تا اون هدف ارضا شود.

Partial order planning: پلن هایی که تا الان دیدم یک برنامه کامل بود که این اکشن را انجام بده بعد اون اکشن ولی در این روش میگوییم مثلا حتما لازم نیست بعضی اکشن ها قبل اکشن دیگر رخ بدهند بلکه میتوانند به صورت موازی اجرا شوند یا ترتیب زیاد مهم نیست. بر خلاف قبلی ها که تو فضای حالت میگشتیم، به حالت مسئله کاری ندارد به فضای پلن ها نگاه میکند میاد به پلن در هر مرحله یک مقداری اضافه میکند تا بتواند جواب بدهد. به این صورت هست که اکشن الف باید قبل از اکشن ب صورت بگیرد چون لیترالی را تولید میکند که جز پیش شرط اکشن ب و جز effect اکشن الف هست.

فصل یازدهم:

عامل چطوری با عدم قطعیت برخورد میکند یعنی چطور میتواند استنتاج احتمالی انجام بدهد در حالتی که دانش آن قطعی نیست؟ دقت کن عدم قطعیت از عدم دانش کافی میاد که با احتمال نشان میدهیم. فضای نمونه میتواند متناهی نباشد و پیشامد میشود هر زیر مجموعه از فضای نمونه که event گفته میشود. پیشامد ها مجموعه هستند و میتوانیم اعمال مجموعه را روی اینها اجرا کنیم. P(s)=1 یعنی وقتی تاس ما 6 تا وجه دارد اگر هر کدام از اینها آمد ما باید یک احتمال براش در نظر بگیریم. P(a) احتمال رخ داد هر عضو از مجموعه باید بیشتر از 0 باشد. اگر رخداد یکی باشد احتمال رویداد ها را میتوانیم حساب کنیم.

فصل یازدهم:

عامل ها در این فصل میخواهند با توجه به ادراکی که دریافت میکنند احتمال وقوع اتفاقات مختلف را بدست بیاورند تا نسبت به آن بتوانند معیار کارایی خود را بیشینه کنند. از احتمال برای مواجهه با عدم قطعیت استفاده میکنیم. یک مشکلی که در فصل 4 داشتیم این بود ما تمام حالات ممکن را در مجموعه حالات باور ذخیره میکردیم ولی عامل نمیدانست هر کدام از این حالات چه قدر احتمال دارند که به وقوع بپیوندند؟ و واسه اون شرایط هیچ پلنی نیست که تمام حالات باور را به یک هدف برساند. در نتیجه معمولا استفاده از این رویکرد هیچکاری نمیتوانیم انجام دهیم و به جواب نمیرسیم در صورتی که اگه عامل احتمال بلد بود و میدانست اتفاق افتادن کدام ها بیشتر هست به جواب ها میرسیم. رویکردی که در فصل منطق داشتیم این بود که واسه جنبه های مختلف متغیر های منطقی تعریف میکردیم و دنبال روابط بین اینها بودیم ولی مشکل این هست که به سادگی نمیتوان به این رسید. دانش عامل را به جای منطقی میایم به شکل احتمالاتی نمایش میدهیم و میزان آن احتمالی هست یعنی مثلا دندان طرف درد کرد بخاطر خرابی نیست شاید بخاطر بد خوردن هست. اگر بخواهیم به همه چیز به شکل منطقی نگاه کنیم نمیتوانیم هیچ استدلالی انجام بدهیم.

فضای نمونه که با امگا بزرگ نشان میدهیم که همه نتایج ممکن یک فضای آزمایش هست مثلا واسه تاس 6 تا هست. و به هر کدام از این 6 عدد یک اتمیک ایونت هست. دقت کن احتمال هر کدام از این اتمیک ایونت ها باید بین 0 و 1 باشد و جمع همه آنها بشود 1 و واسه هر احتمالی که میخواهیم میایم احتمال اون مجموعه اتمیک ایونت ها را جمع میزنیم مثلا واسه دیدن عدد زوج تاس میشود 3/6 or ½. دامنه هم همان مجموعه مقادیر هست دیگه. متغیر گسسته و پیوسته داریم ولی با گسسته کار میکنیم بیشتر. دامنه هم محدود یا نامحدود میتواند باشد. متغیر های تصادفی با حروف بزرگ نشان داده میشوند. احتمال هر چیز با نقیض خودش میشود 1. استقلال: برای حساب توزیع joint کافی هست توزیع هر کدام از این 2 را در هم ضرب کنیم. یعنی دونستن یک متغیر دیگر تاثیری روی احتمال متغیر اولی ندارد. استقلال به ما از لحاظ حافظه کمک میکند چون یک جدول 32 تایی را به یک جدول 8 تایی و یک جدول 4 تایی تقسیم کرده ایم چون اب و هوا قطعا به دندان درد ربطی ندارد. این مفهوم فعلی استقلال فعلا به ما کمکی نمیکند چون عوامل کاملا مستقل از همدیگر نیستند پس باید به یک استقلال شرطی برسیم که یک ضلع سومی هم وجود دارد که ما اگر آن را بدانیم دو متغیر ما مستقل هستند از همدیگر.

فصل دوازدهم:

استنتاج احتمالی: معرفی شبکه بیز: اگر ما برای چند متغیر تصادفی تابع joint اینها را داشته باشم میتوانیم هر احتمالی را حساب کنیم ولی مشکلی که هست این هست که این جداول خیلی بزرگ هست. اون استقلال شرطی یعنی اینکه اگر دندان من پوسیدگی داشت دیگر دندان درد ربطی به متغیر catch ندارد. حروف بزرگ یعنی خود متغیر تصادفی و شروع با حروف کوچک یعنی مقدار اون. کلا استقلال شرطی یعنی دانستن توزیع شرطی معلوم به شرط علت برای ما کافی هست و اون یکی را انگار نمیخواهیم. یعنی ترافیک و چتر معلوم همان باران باریدن هستند. یعنی ما فقط نیاز داریم احتمال فرزند به والد را بدانیم به شکل گرافیکی. شبکه بیز یک dag هست. گره ها همان متغیر های تصادفی هستند یال ها عملا نشان دهنده تاثیر مستقیم بین گره ها هستند که این همان مفهوم locality در شبکه بیز هست. مثلا بین چتر و ترافیک رابطه مستقیمی نیست چون دلیل مشترک فقط دارند. به ازای هر گره جدول احتمال را هم ذخیره میکنیم و با کمک این جداول احتمال شرطی را تعریف میکنیم. جدول هر راس هم توزیع مربوط به اون گره به شرط parent هاش. و اعداد این جدول به ما نشان میدهند که یک راهی برای محاسبه توزیع joint هست یعنی بیا مقدار فعلی را با parent ها ضرب کن تا این توزیع بدست بیاد. دقت کن اول جداولی که داریم تک متغیر هستند چون هیچکی parent کسی دیگه نیست. ببین توزیع میتواند یکسان باشد و ما لزوما یک ساختار شبکه بیزین نداریم ولی اعداد درون جدول کاملا متفاوت هستند. دقت کن در مثال زنگ زدن هنگام دزدی ترتیب توپولوژیکی نوشتیم برای dag که در شبکه بیزین هم هست. تا اینجا با ساختار شبکه آشنا شدیم پس هر مسئله ای که داشتیم بیاد یک ساختار گرافی بسازد و جداول را تولید و مقدار دهی کند یک شبکه ساخته و ادعا میکند که توزیع joint متغیر ها با ضرب اعداد در جدول بدست میاد حالا سوال این هست که شبکه بیز و ساختار گرافی یکتا نیست چطوری این را تعریف کنیم که واقعا اعداد اون جدول ضرب شوند توزیع joint را میدهند. ترتیب توپولوژیکی این است که گرافی که توش حتما دور نیست جوری گره ها را شماره گذاری کنیم که پدران شماره کمتری از فرزندان بگیرند. شرط دوم هم بحث استقلال شرطی هست. هر گره به شرط پدرش از بقیه مستقل هست. توزیع حاشیه ای یا marginal میایم روی همه متغیر های حذف شده سیگما میزنیم و جمع میزنیم مثلا در مثال اول چون +a حذف شده هست باید اون رو هم حساب کنیم. بعد هم که باید احتمال هر کدام را به شرط پدران آن بنویسیم. در مثال بعدی واسه p a اومدیم از توزیع شرطی استفاده کردیم چون ما توزیع g را داریم توزیع a|g هم داریم پس میتوانیم توزیع a را بیابیم. دقت کن گره فعلی از غیر نوادگان خود مستقل هست. اون 2^n-1 بخاطر این هست که عدد آخر میشود منهای همه و لازم نیست بنویسیم اصلا. دقت کن در شرایط عادی اگر مثبت رو داشتی لازم نیست منفی رو ذخیره کنی منفی از 1 منهای مثبت بدست میاد. توی همین تست هم اگر منظور سطر هایی بود که مستقل نیستند دیگه بعلاوه یک نداشت گزینه 2 میشد.

فصل دوازدهم- استنتاج بخش دوم:

در شبکه بیز با فرض اینکه استقلال شرطی داریم گره ها متغیر های تصادفی هستند و یال ها به طور کلی میشود گفت روابط علی یا تاثیر مستقیم دارند مثلا x دلیل رخ دادن y هست ولی این دقیق نیست مثلا فکر کنید در یک حالتی هیچ رابطه علی نباشند مثلا باران آمدن به طور مستقیم دلیل دیر رسیدن ما به کلاس نیست به همین منظور دقیقا نمیتوان گفت نشان گر رابطه هست پس به طور کلی میتوانیم بگوییم همه اینها به فرم استقلال شرطی هستند و گفتیم که گره ها جز نوادگان آن مستقل هست دقت کن ما میتوانیم انواع استقلال شرطی را از یک ساختار استخراج کنیم. مثلا در مثال همین صفحه میتوانیم بگوییم w و x به شرط دانستن متغیر y از هم مستقل هستند. دقت کن خیلی این بخش ها مهم نیست. در صفحه بعدی هر 3 متغیری که داشته باشیم را شکل اون سیاه بالا میتوانیم بنویسیم. چیزی که از میخواهند این اثبات نیست میگویند که بین این دو گره x y آیا به شرط دانستن مقدار مجموعه متغیر z مستقل هستند یا خیر. به این z و مجموعه شواهد گفته میشود. کار این الگوریتم به این شکل هست که متغیر های سه گانه را باید هم آنالیز میکند اونهایی که با هم رابطه دارند. اولیش زنجیره علی هست که x دلیل مستقیم y و y دلیل مستقیم z هست. و وقتی ما مقدار y را بدانیم که اینجا شاهد ما هست تاثیر مستقیم بین x و z را بلاک کرد. حالا یک شکل دیگر این هست که y دلیل مشترک x z باشد مثلا اینکه پوسیدگی دلیل مشترک دندان درد و گیر کردن هست قبلی هم مثل باران بود و اگر مقدار y را بدانیم دو فرزند مستقل میشوند یعنی اگر بدانیم پوسیدگی داریم دیگر دندان درد هیچ ربطی به گیر کردن ندارد. مورد سوم ساختار v شکل یا تاثیر مشترک که انگار دو متغیر یک فرزند یکسان دارند منتها در اینجا اگر مقدار z را بدانیم دیگر مستقل نیستند و این 2 اگر مقدار این را بدانیم میتوانند روی هم تاثیر بگذارند انگار برعکس قبلی ها هست یعنی اگر ترافیک باشد و بدانیم باران نیامده است آن موقع احتمال مثلا بازی فوتبال بیشتر هست و به نوعی انگار با دانستن z این مقادیر رو هم تاثیر گذاشته اند و ما فهمیدیم از این یکی اون یکی را. یعنی وقتی بدانیم یک چیزی رخ داده که معلول دو چیز دیگر هست و بدانیم یک چیز دیگر که علت هست رخ نداده هست پس اون یکی بوده هست و این دقیقا با 2 تای قبلی در اینجا متفاوت هست و قبلی ها با دانستن بلاک میکرد رابطه را ولی اینجا وقتی شاهد را بدانیم انگار ارتباط شکل گرفت هست. حالا الگوریتم D-separation: 3 تایی به دو دسته فعال و غیر فعال تقسیم میشوند و گره خاکستری یعنی ما مقدار یک گره را بدانیم دقت کن ساختار v شکل در شروع غیر فعال هست ولی وقتی مقدار گره میانی یا از نوادگان گره میانی را بدانیم فعال میشود. این الگوریتم 2 گره میگیریم و مسیر ها را بدون جهت فکر میکنیم بررسی میکنیم مسیر های بین این 2 فعال هستند یا خیر و اگر یک الگو غیر فعال 3 تایی داشته باشیم یعنی کلا این 2 تا هم غیر فعال هست و به ازای تمام مسیر چک میکنیم که روی هر مسیر اون الگو 3 تایی غیر فعال وجود دارد یا نه. ولی اگر وجود نداشت احتمالا وابسته هستند ولی چیزی نمیتوان گفت و باید به مقادیر جدول مراجعه کنیم. دقت کن از اول مسیر شروع کن 3 تا 3 تا بررسی کن بعد یکی بیا جلو. بحث چی هست کلا؟ دنبال این هستیم که توزیع احتمال را برای یک متغیر بدست بیاریم که نام آن را متغیر مورد پرسش یا query میگذاریم به شرط یک سری رخداد با کمک متغیر های شواهد یعنی این e ها رخ داده اند حالا میخواهیم توزیع احتمال q را حساب کنیم به متغیر هایی که جز هیچ کدام از این 2 دسته نیستند میگوییم مخفی یا hidden. دقت کن وقتی q را با حرف بزرگ نوشتیم یعنی دنبال توزیع احتمال یا توزیع joint هستیم. و در گام اول سراغ سطر هایی میرویم که e ها رخ داده اند و گار فالس بودند کاری نداریم در گام دوم میایم به ازای متغیر های مخفی که میگیرند و در این فرمول نیستند اون فرمول معروف سیگما را میزنیم و جمع میزنیم اون متغیر های مخفی حذف میشوند و مخرج بدست میاد یعنی انگار همون سطر ها را جمع زدیم ولی مخرج بدست میاد به این روش استنتاج به کمک شمارش گفته میشود و دقت کن به ازای تمام مقادیر که دو متغیر hidden در مثال بعدی گرفته اند باید توزیع joint b را حساب کنیم و انگار هر 5 تا متغیر را آورده باشیم و این مقدار زیادی میشود و ضرب های بسیار بزرگی هستند و اگر n متغیر بود و r متغیر hidden بود و انگار 2^r تا جمع انجام میدهیم چون به ازای هر حالت متغییر hidden یک ضرب داریم که تهش با هم جمع میشوند و هر کدام از ضرب ها همه متغیر ها را شامل میشوند و مرتبه زمانی بالا میرود در حالتی لزومی ندارد این ضرب ها اینقدر بزرگ شود مثلا p b در همه ضرب ها اومده هست در صورتی که میشود فاکتور گیری کرد. روش حذف متغیر: بیا عمل جمع زدن و توزیع ساختن را یک به یک انجام بده یعنی اول بیا یک توزیع 2 تایی بساز بعد جمع کن بعد با متغیر سوم ادغام کن بعد جمع کن یهو اینقدر بزرگ نشود. و ما در واقع میایم این ضرب ها را به شکل join پایگاه داده 2 جدول به هم اضافه میکنیم که ستون مشترک در اون مقدار مشترک ضرب میشود و وقتی این را بدست آوردیم کار ما تمام نشده بلکه تازه باید از شر متغیر های hidden که در این مثال R,T هستند خلاص شویم. با sum out کردن جمع میزنیم مثل توزیع marginal و حذف میکنیم یعنی انگار مثبت منفی با هم جمع میزنیم. از لحاظ حافظه هم این استنتاج با شمارش مناسب نیست چون جدول خیلی گنده میشود. به همه جدول هایی که استفاده میشود فاکتور گفته میشود. دقت کن با یک ترتیبی این متغیر های hidden را حذف میکنیم و همچنین دقت داشته باش که وقتی join زدی بعدش سریعا متغیر اولی را حذف میکنی با sum out این متغیر جز متغیر های مخفی باید باشد. همین 3 مرحله رو برو.

P b چرا اومده بیرون؟ چون حالت فاکتور گیری دارد و یک چیز مدام تکرار میشود. ما میخواهیم بزرگترین فاکتور را تا حد امکان کوچک نگه داریم. و ترتیب انتخاب متغیر های hidden تاثیر دارند روی این. Event ها را با حروف کوچک مینویسیم. و میخواهیم نشان بدهیم که ترتیب انتخاب hidden event ها خیلی تاثیر دارد روی پیچیدگی روش حذف متغیر. دقت کن در اون مقال اندازه فاکتور اول 2 هست فقط به ازای z مثبت و منفی.

فصل دوازدهم بخش سوم:

روش تقریبی از جواب بهینه یک تخمین میزنند مزیت این هست که خیلی کارا اجرا میشوند و منابع و زمان زیادی ندارند بر خلاف حذف متغیر که در زمان نمایی بود. یک مورد نمونه برداری هست یعنی اینکه بیایم از متغیر های تصادفی نمونه تولید کنیم یعنی یک جورایی اون آزمایش تصادفی را اجرا کنیم و با استفاده از این نتایج تخمین بزنیم. مثلا یک سکه بگیریم به ازای هر بار پرتاب کردن یک نمونه ایجاد میشود و میخواهیم ببینیم چند بار شیر میاد و اگر 7 تا بیاد هفت دهم میشود واضح هست که این یک جواب تقریبی هست. ما روی شبکه های بیز تمرکز میکنیم. و دنبال روشی هستیم که در بینهایت میل کند به اون عدد واقعی که جواب هست. 4 روش داریم. اول اینکه چطور از یک توزیع نمونه تولید کنیم؟ یک روش ساده این هست که از تابع رندوم یک عدد رندوم بگیریم برای شبیه سازی این آزمایش به نوعی که مثلا اگر تا آستانه 0.6 بود این عدد ترو و اگر بیشتر بود این فالس هست پس برای هر 4 روش به این گام نیاز داریم برای نمونه برداری. Prior sampling کلا به شواهد کاری ندارد و نمونه ها را به شیوه قبلی که گفتیم تولید میکند. این روش به ترتیب توپولوژیکی از شبکه بیز عمل میکند و شروع میکند. و مثلا میگوییم اون عدد بین 0و1 اگر کمتر از نیم بود +c و اگر بزرگتر از نیم بود -c باشد. و برای گره بعدی که s هست میایم دوباره همون شیوه رو اجرا میکنیم و یک عددی بین 0 و1 انتخاب میکنیم طبیعتا 0.9 بیاد منظور ما -s هست. و برای نمونه بعدی ها دقیقا همین عملیات را تکرار میکنیم مثلا قبلی +c بود این سری -c رخ بده مثلا و بعد کل اون نمونه را return میکنیم. تخمین هم اینطوری بدست میاد که مثلا +w 4 بار رخ داده هست پس میشود 4 بر روی کل نمونه ها. احتمال شرطی هم میتوان حساب کرد یعنی توزیع متغیر c به شرط +w حساب کنیم پس وقتی مثبت w هست دیگر منفی ها را در نظر نمیگیریم چون کمکی نمیکند. واسه آخری هم چون هیچ سطری نداریم که هر دو تا منفی باشند پس نمیتوانیم جواب بدهیم. باید نشان بدهیم این روش سازگار هست یعنی اگر این را بی نهایت بار نمونه تولید کنیم باید احتمال بدست آمده از این روش میل کند به احتمال واقعی متغیر ها. Rejection sampling: به شواهد بر خلاف قبلی اهمیت زیادی میدهد و فقط نمونه هایی را تولید میکند که سازگار هستند با evidence های احتمال. یعنی بر خلاف قبلی میگه فقط اونایی که +s دارند را تولید کن -s که بدرد ما نمیخورد و کار را ادامه نمیدهد و دور میریزد. این روش نسبت به قبلی کارا تر هست و وقت را تلف نمیکند. مسئله این هست که در این الگوریتم باید کلی نمونه ها را دور بریزیم و این خوب نیست. راه حل این هست که شواهد را ثابت فرض کنیم. دقت کن این فقط برای متغیر های شواهد هست. و هر چه قدر که یک نمونه وزن کمتری داشته باشد یعنی اهمیت کمتری دارد. و از بقیه متغیر ها نمونه گیری میکنیم و فقط به شواهد رسیدیم وزن آن را در احتمال شرطی ضرب میکنیم. این روش خوبی هست منتها به اون متغیر هایی کار دارد که بعد از متغیر های شواهد آمده اند ولی با اونایی که قبل اینا اومدن اصلا کاری ندارند. Gibbs sampling: با تکرار زیاد این فرآیند نمونه های ما به توزیع joint نزدیک تر میشوند و به احتمال واقعی نزدیک تر میشوند. روش: از یک نمونه کاملا دلخواه شروع میکنیم فقط باید با شواهد سازگار باشد یعنی مثلا حتما باید +r را داشته باشد و میاد با توزیع c به شرط بقیه متغیر ها یک نمونه از c تولید میکند و غیر از شواهد برای بقیه هم همین کار را تکرار میکند. و با بینهایت انجام دادن این کار میل میکند به توزیع joint واقعی. و این شرطی که مینویسیم در واقع میخواهیم توزیع احتمال را بدست بیاوریم. هر متغیر را به شرط بقیه نمونه ها نمونه بگیری یعنی میخواهی توزیع را بدست بیاوری. دقت هم کن از اون شیوه دقیقا استفاده میکنی که عدد رندوم میدهی اگر 0 تا 0.7 بود +s و اگر نبود -s هست. و دقیقا همین کار نمونه برداری را برای بقیه هم بکن. و این کار را اگر زیاد انجام بدهی میل میکند به توزیع واقعی. دقت کن در شبکه بیز ما p یک متغیر به شرط بستگان آن را داریم ولی سایر متغیر ها را نداریم پس باید دنبال راهی باشیم که بتوانیم اینها را حساب کنیم. از فرمول احتمال شرطی باید بریم و در صورت توزیع joint قرار میگیرد خروجی این کار این هست که ما تمام توزیع هایی که میخواهیم در جداول CPT اینها هستند. پس کافی هست برای محاسبه این کسر این جداولی که s در آنها وجود دارد را join کنیم. در الگوریتم سوم برای بررسی وزن یک نمونه میایم وزن را ضرب میکنیم در احتمال متغیر شاهد به شرط parent هاش. 1 وزن اولیه هست. در الگوریتم چهارم ترتیب میدهند و میگویند کدام دنباله ممکن هست رخ بدهد یکی از روش ها این هست که برو ببین شواهد تغییر کرده اند یا نه اگر تغییر کرده باشند نمیتواند رخ بدهد همچنین اگر در هر مرحله یک علامت عوض شود موردی ندارد. در توزیع احتمال در کسر تست بعدی میشود هر متغیر به احتمال parent هاش. مخرج هم که فقط متغیر hidden را اضافه میکنی به ازای اون جمع میزنی.

مفاهیم یادگیری ماشین:

تا الان یک مدل را به ما میدادند و ازش استفاده میکردیم و به جواب میرسیدیم مثلا شبکه بیز را میدادند با CPT و ما استفاده میکردیم و به جواب میرسیدیم یا گراف میدادند میگفتند به جواب برس ولی الان میخواهیم با داشتن یک سری نمونه یک مدل بسازیم. یعنی از روی دیتا مدل را بسازیم. حالا چرا مهم هست؟ ممکن هست هر چه قدر الگوریتم خوبی داشته باشیم ولی کامپیوتر با داده ای روبرو شود که نداند چه کاری بکند پس خود کامپیوتر بتواند تصمیم بگیرد بهتر هست. یک سری مسائل هستند برنامه نویسی برای آن ممکن نیست و به شکل نا خودآگاه هست پس بهتر هست خود ماشین یاد بگیرد تا این کار را بکند البته میتوانیم خود یادگیری را به شکل استنتاج احتمالی در بیاریم که میشود دید بیزی به یادگیری. انواع الگوریتم داریم. در این درس الگوریتم های supervised را بررسی میکنیم. در این الگوریتم داده ای که در اختیار ما قرار میگیرد به شکل n تا ورودی و خروجی هست. به شکل زوج مرتب. و کامپیوتر با آنالیز اینها باید داده های بعدی را بررسی کند. به خروجی میگوییم برچسب یا label ولی در unsupervised حتی این مورد را هم نمیدهیم. دقت کن هر خروجی توسط یک تابع f x تولید شده است ولی اول ما نمیدانیم و سعی میکنیم یاد بگیریم. به این تابع یک فرضیه گفته میشود. و باید یک h یی پیدا کنیم که بتواند اون f را تخمین بزند. مجموعه توابع که فرضیه هستند میشود فضای فرضیه که بین اونها یکی را برمیداریم. این مسئله تشخیص اسپم بودن ایمیل یک نوع مسئله دسته بندی هست یعنی یک سری ورودی که در اختیار داریم به کدام کلاس تعلق میگردد. هدف گروه بندی هست مجموعه هم یک تعدادی داده به همراه برچسب های آنها هستند. برای ورودی های مسئله همیشه یک تعدادی ویژگی تعریف میکنیم و ورودی را بر اساس آنها نشان میدهیم. مثلا اگر کلمه free را داشت 1 اگر نداشت صفر یا مثلا حروف بزرگ هست یا نه یا علامت دلار هست توش یا نه. و هر ایمیل نشان داده شده با یک بردار نمایش داده میشود و بر اساس اینکه مقادیر این فیچر ها چه قدر هست تصمیم گیری میکنیم. و هر ورودی تبدیل میشود به یک برداری از ویژگی هایی که تعریف کردیم. الگوریتم naïve bayes: این یک روش مبتنی بر مدل هست. این یعنی الگوریتم میاد یک مدل از اینکه جهان چطور کار میکند میسازد و بر اساس آن استنتاج میکند. پس هدف این هست که مسئله ای که میدهند را مدل کنیم به صورت یک شبکه بیز. و در این شبکه بیز گره ها ما میشوند اون فیچر های ما و یک گره هم داریم برچسب نهایی یا خروجی هستند که بهش y میگوییم و هر کدام از اینها متغیر تصادفی هستند چون از قبل نمیدانیم مقدار چی هست و قطعی نیست بعد فقط کافی هست ساختار شبکه بیز این گره ها را بکشیم و مقادیر احتمال در CPT ها یعنی خودش به شرط parent هاش رو داشته باشیم یادگیری ما حل هست چون هر احتمالی را میتوانیم حساب کنیم. و برچسب با احتمال بیشتر را به عنوان خروجی بر میگردانیم. چرا رفتیم سراغ شبکه بیز؟ چون جدول توزیع توام خیلی بزرگ خواهد بود. اینکه چه راسی به چه راسی یال دارد و مقادیر CPT ها به چه شکل هست نیز باید بفهمیم به چه شکل باشد چون کسی به ما این را نمیدهد. الگوریتم naïve bayes میاد جواب سوال اول را میدهد میگوید ساختار گراف به همین شکل باید باشد اون رابطه استقلال شرطی فصل قبل رو یادت بیار میگوید به شرط دانستن y ما گره ها از هم مستقل هستند. دقت کن f ها هم بهم یال مستقیم نباید داشته باشند. این سوال جواب اول حالا باید سراغ جواب دوم و اعداد داخل این جداول برویم. با این 2 میتوانیم احتمال ها را بررسی کنیم روی ایمیل ها. و میگیم اون y را انتخاب کن که این احتمالی که داریم را بیشینه کند. در واقع دنبال y هستیم که این کسر را بیشینه کند و دقت کن چون مخرج کسر چون یکسان هست همیشه و اصلا y ندارد اصلا تو مقایسه ما نمیاد پس صورت باید بیشینه شود پس باید احتمال joint را بنویسیم. حالا باید به سوال دوم جواب بدهیم که احتمال داخل CPT ها را بتوانیم حساب کنیم. این مقادیر چی هستند؟ مثلا اگر این ایمیل HAM باشد چه قدر احتمال وجود دارد که این کلمه را دیده باشی چون فقط به ما یک سری ایمیل میدهند. تخمین پارامتر: از دیتا میخواهیم یک سری پارامتر تتا را تخمین بزنیم. یک روش این هست که بیا تتا را مقداری نسبت بده که بیشترین احتمال را به نمونه ها نسبت بدهد یعنی تتا را جوری بزار که احتمال این چیزهایی که داریم بیشینه شود. حالا برای پیدا کردن تتا بیایم اون تابع رو بر حسب تتا بنویسیم.

دو تای اول میشود تتا و سومی میشود 1 منهای تتا. بعد باید مشتق بگیریم از اون تابع. این موارد را با یک سری فرض نوشتیم یعنی گفتیم مهره اول حتما قرمز هست یعنی فرض کردیم همه نمونه ها از یک توزیع ثابت انتخاب شده اند. فرض دوم این بود که این 3 آزمایش x1 x2 x3 از هم مستقل هستند و به ما کمک کرد چون توانستیم احتمال joint را حساب کنیم با ضرب اینها با هم. آزمایش دو جمله ای خودش n باز آزمایش برنولی هست. پرتاب سکه هم به همین شکل هست یعنی برنولی هست. لگاریتم یک تابع صعودی اکید هست. دقت کن خود حاصل قبل مشتق سخت هست پس لگاریتم میگیریم بعد مشتق میگیریم که ساده تر میشود. دقت کن حاصل اون تتا ضربدر تتا ضربدر 1 منهای تتا را بیشینه کنیم. کلا از تتا اون نتیجه رنگ طلایی را در ذهنت داشته باش یعنی در واقع دنبال تعداد رخداد هایی یک چیزی در یک کلاس بر روی کل رخداد های اون کلاس هستیم. منتها موردی که دارد این هست که: که باعث overfitting میشود یعنی در مجموعه تمرینی دیده هر موقع کلمه دقیقه بیاد جز spam حساب میشه بعد نسبت به داده های جدید هم این را جز spam قرار میداد چرا چون حاصل اون احتمال صفر میشد پس کل احتمال را 0 میکرد و جز spam میشد. در روش LaPlace smoothing برای حل این مشکل اومده هست که میایم هر خروجی را به شکل الکی k بار بیشتر دیده ایم. یعنی به خروجی یک مقداری اضافه میکنیم این باعث میشود دیگر احتمال 0 ظاهر نشود. انگار مثلا در سکه یک شیر و یک خط بیشتر اضافه میکنیم. و اگر این k به سمت بینهایت برود احتمال های ما مساوی میشود. جواب تست: عین همون احتمال صفر شدن ham و اینکه تمام کلمات در spam دیده شده بود که میشد spam و مشکل overfitting بود. درخت تصمیم: یکی دیگر از الگوریتم های یادگیری ماشین هست، روش قبلی یک روش مبتنی بر مدل بود یعنی میگفت داده ها بر اساس این مدل تعریف شده اند بعد میومد تخمین میزد اون پارامتر را یعنی در واقع دنبال اون مدل و توزیعی بود که داده ها از اون تولید شده اند اما در این روش اصلا به توزیع کاری ندارد و صرفا رابطه بین ورودی و خروجی را کشف میکنند. و درخت تصمیم میاد بردار فیچر ها را مپ میکند به اون خروجی. در واقع یک نمایشی از یک تابع هست که مپ میکند. هر نود داخلی نماینده یک تست روی مقدار یکی از فیچر ها هست و شاخه هایی که ازش خارج میشوند با مقادیر ممکن این فیچر ها علامت زده شده اند و برگ های درخت مقادیر ممکن خود تصمیم هستند. زمانی که در یک دسته همه ورودی ها برچسب یکسان بگیرند دیگر آن شاخه را ادامه نمیدهیم مثلا ورودی p none باشد کلا میگیم نه و دیگه ادامه نمیدیم. اونایی هم که some هستند چون بله شده اند دیگر ادامه نمیدهیم چون نتیجه تصمیم مشخص هست اینقدر میرویم تا همه نتایج تصمیم مشخص شود. وقتی درخت را ساختیم به ازای ورود داده های جدید تست میکنیم تا به yes برسیم و همیشه دنبال درختی هستیم که عمق کمتری داشته باشد چون تست کمتر دارد و سریعتر به جواب برسیم. برای اینکه درخت ما کوچک تر باشد باید فیچری را انتخاب کنیم که متمایز کننده ترین باشد. دقت کن اگر در یک دسته ای تصمیم همه برچسب یکسان داشته باشد اون خروجی تمام هست و انگار تو عمق کمتری به جواب میرسیم. ولی مثلا بر اساس دسته تایپ همه رو بازم باید تست انجام بدهیم و هیچی تمام نشده چون دوست داریم سبز ها یکجا و قرمز ها یکجا باشن تا سریعتر به جواب برسیم.

ساختار درخت تصمیم یک ساختار بازگشتی هست چون میخواهیم دیتا ها را دسته بندی کنیم اما تا کجا؟ 4 حالت ممکن هست رخ دهد: 1. همه نمونه ها برچسب یکسان داشته باشد و کار تمام هست. 2. ممکن هست برچسب ها متفاوت باشد و باید کار را ادامه دهیم. 3. هیچ نمونه ای توی اون دسته نباشد و کار تمام هست. 4. در یک دسته ای نمونه های بله و نه داریم ولی هیچ فیچر دیگه ای باقی نمانده که اینها از هم جدا شود یا هر چی بزنیم اینها باز از هم جدا نمیشوند هم باید کار را تمام کنیم چون هر فیچری امتحان میکنیم از هم جدا نمیشوند مقدار تصمیم آنها هم متفاوت هست یکی بله هست یک خیر در این حالت هم باید کار را تمام کنیم این زیاد معنی نمیدهد ولی جز noise داده های ما هستند. درخت تصمیم از مجموعه تمرینی ساخته میشود و درصد خطا روی داده های training set مساوی 0 هست. تنها حالتی که ما داده بدیم به این درخت مساوی اون تصمیم نشود همون noise های داده های ما هستند تنها حالت خطا همین هست. ترتیب انتخاب فیچر ها برای ساخت درخت تصمیم: خوب هست با فیچری انتخاب کنیم که متمایز ترین باشد یعنی بتواند نمونه ها را خوب جدا کند تا عمق درخت کمتر شود حالا از لحاظ ریاضی به چه شکل هست: آنتروپی در نظریه اطلاعات هست برای یک متغیر تصادفی تعریف میشود و قرار هست معیاری باشد برای عدم قطعیت برای این متغیر حالا هر چه قدر ما در مورد نتیجه یک آزمایش تصادفی عدم قطعیت بیشتری داشته باشیم آنتروپی بیشتر هست و هر چه قدر بتوانیم بیشتر پیش بینی کنیم اون متغیر تصادفی را آنتروپی کمتر هست مثلا اگر یک سکه بندازیم همیشه شیر بیاد هیچ عدم قطعیت ندارد چون از قبل میدانیم. واحد آنتروپی را با بیت نشان میدهیم یعنی برای نشان دادن این آزمایش به 1 بیت نیاز داریم یعنی 1 برای شیر و 0 برای خط مثلا. یا مثلا برای تاس 4 وجهی میشود 2 بیت. حالا تعریف ریاضی: احتمال هر کدام از اینها را ضرب کن در لگاریتم یک بر روی همین احتمال که خروجی میشود همین آنتروپی. هر موقع نوشتیم B(q) یعنی این فرمول مد نظر ماست. حالا بخواهیم یک متغیر تصادفی تعریف کنیم که بخواد مقدار تصمیم یک نمونه از نمونه های مجموعه تمرینی را نشان بدهد پس مقدار آنتروپی میشود مثبت بر روی کل نمونه های که جمع مثبت و منفی هست. و اگر مثلا فقط مثبت زیاد باشد منفی کم 99 تا آبی 1 قرمز آنتروپی خیلی کم میشود چون با احتمال بالایی میدانیم آبی ممکن هست بیاد. کاری که در درخت تصمیم میکنیم این هست که بر اساس یک فیچر دسته بندی میکنیم. H همان آنتروپی هست و وقتی ما داخل جایی هستیم که فقط آبی افتاده هست h میشود صفر چون مشخص هست فقط آبی میاد در نتیجه هر فیچری که بررسی میکنیم به ما یک مقداری اطلاعات واسه خروجی نهایی میدهد یعنی هر وقتی بر اساس یک فیچر نمونه ها را تقسیم بندی میکنیم اون مقدار آنتروپی کاهش پیدا میکند یا همون خروجی ما واسه متغیر تصادفی. به این اطلاعات information gain فیچر a هست هر چه قدر این بیشتر باشد متمایز کننده تر هست پس میایم فیچری انتخاب میکنیم این info بیشتری داشته باشد. چرا در مرحله اول آنتروپی 1 هست؟ چون مقدار سبز و قرمز مساوی هستند. وقتی info صفر شود یعنی این فیچر اصلا هیچ اطلاعاتی به ما نداد و بدرد بخور نبود.

فصل سیزدهم:

تصمیم گیری های پیچیده: در این فصل بر میگردیم به فصل های ابتدایی و بهبود میدهیم عامل در فصل های اول منطقی بود و دنبال این بود سریع به یکی از اهداف برسد و محیط قطعی بود. و تصمیم میشود اون گام هایی که تو تعداد کمتر به حالت هدف برسیم. دو نکته اینجا تفاوت دارد 1. محیط تصادفی هست و نتیجه اکشن از قبل مشخص نیست. 2. سودمندی عامل فقط بودن در هدف نیست بلکه در هر لحظه به ازای هر تصمیم یک مقداری پاداش دریافت میکنیم. و هدف این هست که در حالت مسئله بیشترین سودمندی حاصل کدام اکشن هست. در هر جهتی که بخواهم حرکت کنم 80 درصد احتمال دارد به این سمت حرکت کنیم و اون مسیر هایی که عمود بر آن هستند هر کدام 10 درصد میگیرند پس دیگر یک پلن ثابت نداریم. سودمندی در یال ها وجود دارد در این مسائل نه در حالت ها دقت کن. یعنی ما تصمیم هایی میگیریم که در کل سود بیشینه را به ما بدهند. قبلا تابع ریزالت تابع انتقال ما بود و نتیجه را در آن نمایش میدادیم ولی اینجا چون محیط قطعی نیست و ممکن هست هر اتفاقی بیفتد پس تابع انتقال باید به صورت احتمالاتی باشد. یعنی میخواهیم بگیم به چه احتمالی با این اکشن این نتیجه ممکن میشود که با p نشان میدهیم. و برای هر کدام از این تابع ترنزیشن باید پاداش هم مشخص کنیم مثلا این اکشن اینقدر پاداش دارد. رفتار عامل را با دنباله محیط نمایش میدهیم که همان دنباله اکشن ها یا حالت هایی هستند که به آنها رفتیم. اما تاثیر مارکوف چیه این وسط؟ میگوید اینکه در لحظه بعدی در یک حالت دیگر باشی به این بستگی دارد که فقط و فقط الان کجا هستی و چه اکشنی انتخاب میکنی نه به قبلی ها. یعنی اینکه در لحظه بعدی در خانه 3 باشی مستقل هست از اینکه چطوری به خانه 2 رسیدی الان در خانه 2 هستی و در لحظه بعد میخواهی خانه 3 باشی. کلا هر وقت مدل میاریم یعنی دنبال ساده سازی هستیم. در واقع سیاستی میخواهیم داشته باشیم که در هر حالت مشخص کند چه کاری باید بکنیم و به قبلی ها کاری نداشته باشد یعنی یک تابع هست از حالت به اکشن. سیاست بهینه چیست؟ سیاستی هست که بیشترین سودمندی مورد انتظار را به ما بدهد. یعنی به طور متوسط بیشترین سودمندی را به ما بدهد مثل فصل 5 بازی ها اما بر خلاف قبلی که واسه یک حالت میگفتیم اینجا واسه همه حالات مسئله میخواهیم انتخاب بهینه را انجام بدهیم. چرا در مثال بعدی مستقیم بالا نمیرویم؟ چون به ازای ده درصد ممکن هست در خانه منفی 1 برویم و سود خیلی منفی شود ولی اونجا که میریم تو دیوار درست هست میریم تو دیوار ولی به احتمال 10 درصد ممکن هست بریم بالا واقعا پس این بهتر و هزینه اکشن خیلی کم هست بهتر هست تا اینکه -1 نمره منفی بگیریم. اون living reward هر چه قدر کمتر باشد ریسک کمتری دارد. ولی مثلا خیلی بزرگتر باشد یعنی ریسک را کنار گذاشته حتی با وجود اینکه ممکن هست به منفی 1 برود چون با هر اکشن هزینه را هم زیاد کرده است تا بتواند سریعتر به منفی 1 برسد. ولی مثلا در منفی 2 اینقدر اوضاع بد هست میخواهیم بپریم تو آتیش سریع چون هر کار دیگه ای بکند به ازای منفی 2 واحد سود منفی میگیرد. به ازای هر ترنزیشن یک پاداشی دریافت میکنیم یک روش منطقی این هست که بیایم همه پاداشی که گرفتیم را جمع کنیم اما اینکه صرفا مجموع در نظر بگیریم شاید زیاد مناسب نباشد چون بهتر هست که مقادیر دریافت پاداش بیشتر را زودتر دریافت کنیم مثلا بخاطر عوامل مختلف برای هندل کردن این موضوع میایم یک هزینه تخفیف در نظر میگیریم و هر پاداشی که گرفته باشیم را ضربدر یک مقداری کمتر از 1 میکند و در هر لحظه سودمندی ما گاما برابر میشود و مزیت های زیادی دارد. در مثال بعدی مثلا دنبال این هستیم که برویم خونه سمت چپ به سمت خونه 10 تا از همانجا با سود 10 خارج شویم. دقت کن در این مسائل میتوانیم حالت نهایی اصلا نداشته باشیم یعنی تا ابد بازی ها ادامه پیدا کنند و به همین علت میتوانیم دنباله پاداش نا متناهی داشته باشیم و سود مثبت بینهایت میشود مشکل این هست که سیاست های دیگه هم به همین علت مثبت بینهایت میشود و ما دیگر نمیتوانیم سیاستی اتخاذ کنیم یک راه حل این هست که ما بیایم یک زمان ثابتی در نظر بگیریم یعنی تا ثانیه 100 بگیم به اکشن ها پاداش میدهیم و بعدش دیگه اهمیتی ندارد برام و بر اساس اون سیاست ها را انتخاب میکنیم که بهش راه حل افق محدود گفته میشود. راه حل دوم این هست که از یک ضریب تخفیف بین 0 و 1 انتخاب کنیم چرا چون مقدار سودمندی دنباله ها محدود میشوند حتی اگر دنباله نامحدود باشد یک کران بالای محدود داریم. چرا چون یک دنباله هندسی نزولی میشود که به بینهایت میل میکند. حتی اگر تعداد آن نامحدود باشد. یعنی سیاست های درست هست که تا بینهایت پاداش میدهند ولی با این تخفیف این مشکل را حل میکنیم چون هر لحظه پاداش در هر لحظه که میگیریم چون ضرب میشود در یک عدد بین 0 و 1 هعی کمتر میشود. حالت جاذب هم به این شکل هست که همه سیاست ها به حالت نهایی برسند که البته زیاد استفاده نمیشود و همون تخفیف مورد استفاده هست. رویکرد مشکلات مینی مکس چی بود؟ هر چه قدر عمق را افزایش بدهیم اندازه درخت به شکل نمایی رشد میکند و مشکل بعدی وجود حالات تکراری در مسئله هستش. برای حل مشکل دوم این هست که مثل برنامه نویسی پویا واسه مقادیر تکراری یکبار محاسبه کنیم. مشکل نا محدود بودن هم مثلا بیایم تا یک عمق مشخصی انجام بدهیم بعد از آن بیایم عمق را یک به یک اضافه کنیم تا سودمندی که بدست میاد تغییرات خیلی کوچک باشند چرا چون ضریب تخفیف کمتر از 1 داریم و هعی ارزش کمتری پیدا میکند اون پاداش واسه همین حالات بعدی در عمق بیشتر تاثیر کمتری روی ریشه دارند. در صفحه بعدی: مقدار سودمندی را با v\*(s) نشان میدهیم که مثل تعریف مینی مکس در درخت بازی هست که میگوید مینی مکس یک گره یعنی مقدار اون سودمندی که بدست میاد به شرطی که بعد از اون همه انتخاب ها بهینه باشد. چون بازی شانسی هست پس سودمندی مورد انتظار باید باشد. مقدار سودمندی حالت شانس هم مثل گره شانس در مینی مکس هست یعنی بعد از این اکشن اگر همه اکشن ها بهینه باشد چه حالتی رخ میدهد. ببین اگر اکشن ها قطعی بودند تا میگفتیم سمت راست میرفتیم سمت راست واسه همین هست همه خونه ها مثبت 1 نیستند چون ممکن هست یهو به 1 نرسی بیفتی تو خونه پایینی نحوه محاسبه این اعداد هم توسط الگوریتم هایی که میخوانیم هست. روابط بازگشتی در اینجا تقریبا شبیه به همان محاسبه امید به ریاضی هست یعنی ببین تمام یال هایی که از گره مورد نظر خارج میشود را احتمال را ضرب کن در آن سودمندی که قرار هست بهت برسد r همان پاداش هست که به ازای اون ترنزیشن بدست میاوریم. دقت کن مقدار پاداش باید در ضریب تخفیف ضرب شود این فرقش با امید ریاضی بازی ها هست. و با این رابطه بازگشتی v\* فعلی را از قبلی ها بدست میاوریم. کلا این رو در نظر داشته باش قرار ماکسیموم بگیری بین اون یال هایی که خارج میشود هزینه هم که گفتیم چه شکلی هست. برای حل این رابطه بازگشتی باید از جستجو عمق محدود استفاده کنیم. که یک حدی مشخص میکنیم و میگوییم کلا k تا حرکت دیگر برای ما باقی مانده هست. k=0 یعنی بازی بدون اینکه شروع شود به اتمام برسد. 0.9 ضریب تخفیف هست.

دقت هم کن دنبال رابطه ای بازگشتی هستیم که از اون طریق بتوانیم v1 ها را از روی v0 ها بسازیم. که عین بلمن هست این رابطه منتها یک ضرب احتمال هم داریم که با t نمایش میدهیم. و مقادیر VK را هر چه قدر به سمت بینهایت ببریم این VK با VS که همان سودمندی خود ما هست متمایل میشود یا نزدیک میشود. و با استفاده از این رابطه بازگشتی V0 ها را همه صفر میگذاریم بعد سعی میکنیم V1 ها را با این رابطه بازگشتی حساب کنیم و هر چه قدر این رابطه را تکرار کنیم به V\*S متمایل شود چون عمق زیاد میشود و حالات کمتر و تاثیر کمتر و به صفر میل میکند و به V\* متمایل میشویم. در مثال صفحه بعدی چون به احتمال نیم میرویم به warm و با احتمال نیم میرویم cool پس 2 تا احتمال باید بنویسیم و بین این 2 ماکسیموم بگیریم. تعریف q\* چی بود؟ اکشن a را انجام دادی و بعد از این میخواهی بهینه رفتار کنی کلا سودمندی که بدست میاری به طور متوسط چه قدر هست.