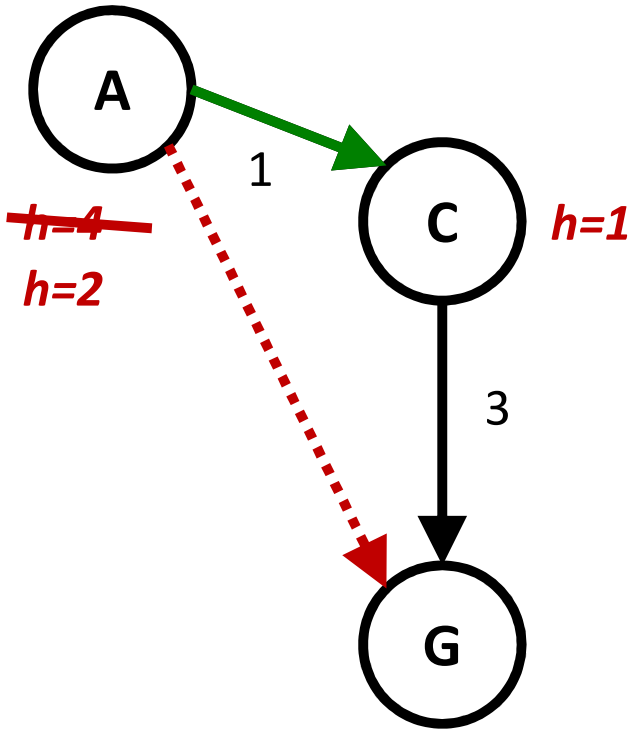
Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

Consistent heuristics

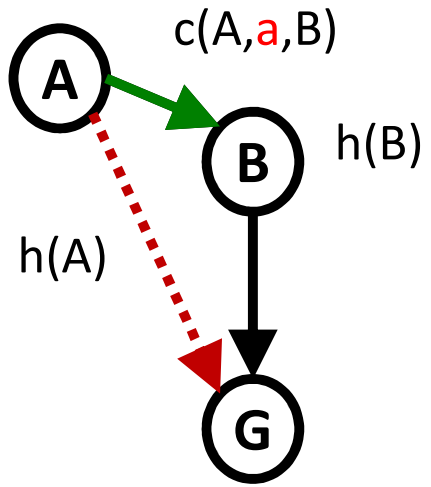
A heuristic is consistent if for every node A, every successor B of A generated by any action a,

$h(A) \le c(A,a,B) + h(B)$

If h is consistent, we have

$$f(B) = g(B) + h(B)$$
$$= g(A) + c(A,a,B) + h(B)$$
$$\ge g(A) + h(A)$$
$$\ge f(A)$$

$f(B) > f(A)$



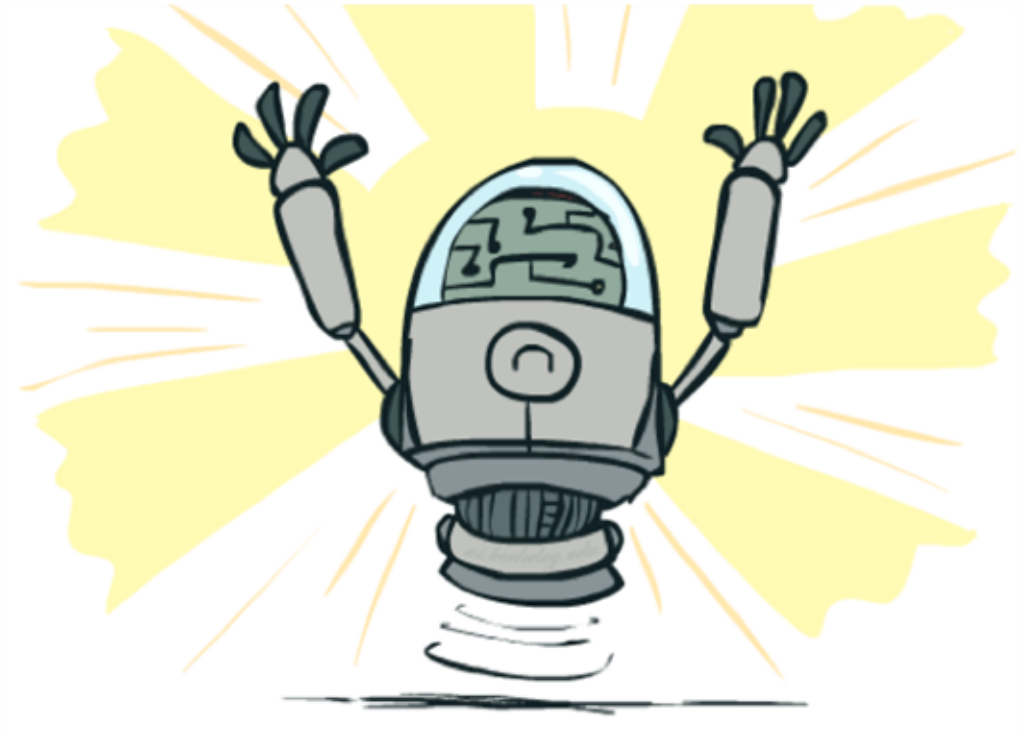
i.e., f(A) is non-decreasing along any path.

Thus: If h(A) is consistent, A* using a graph search is optimal

The first goal node selected for expansions is an optimal solution

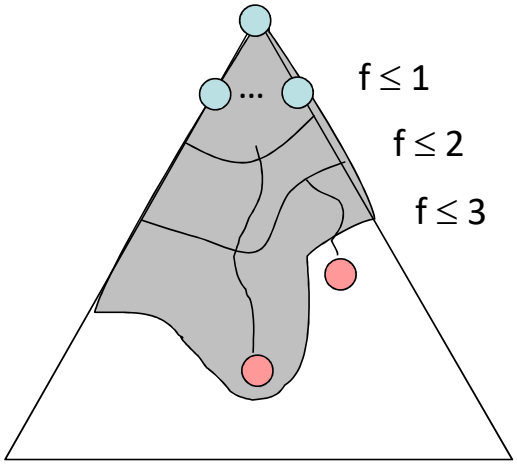
---

Optimality of A* Graph Search



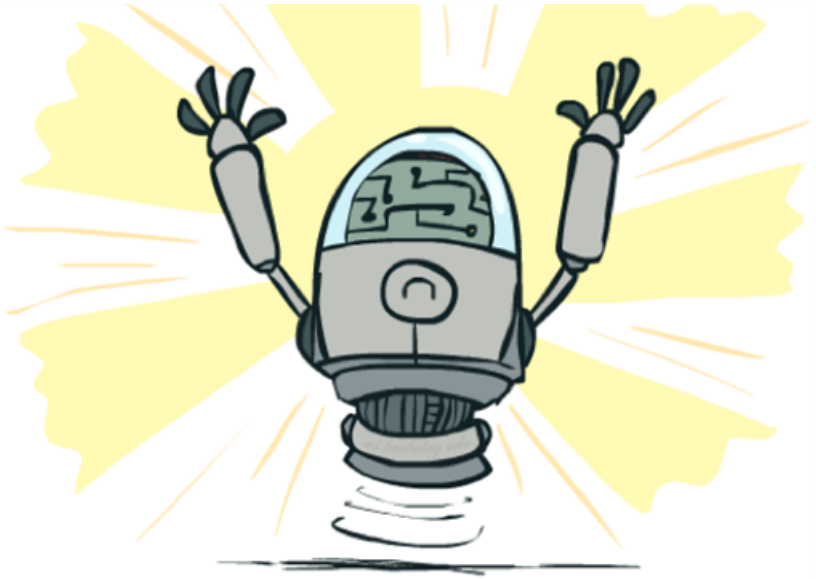
Optimality of A* Graph Search

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal



Optimality

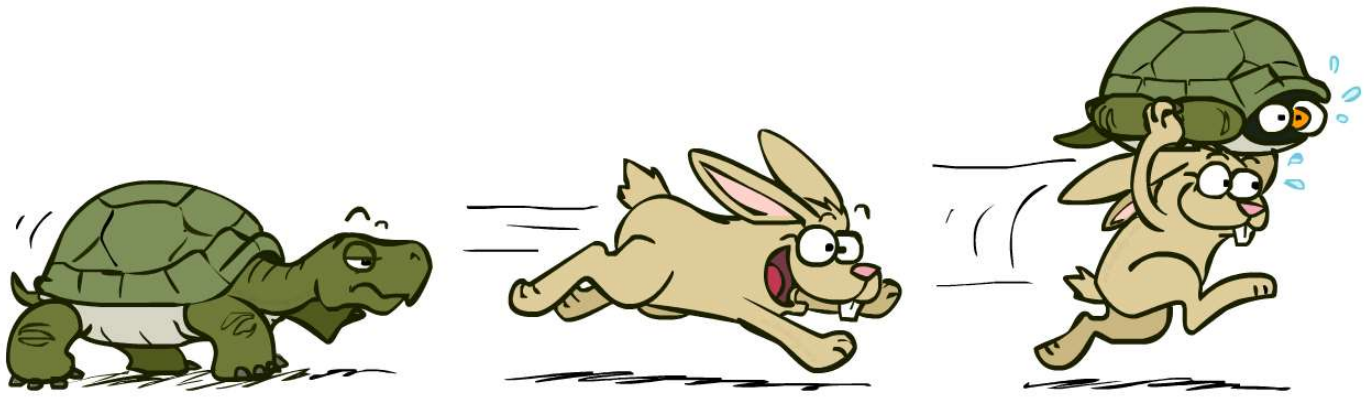
- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



A*: Summary

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



Tree Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Graph Search Pseudo-Code

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

SOME MORE POINTS

Tree Search vs. Graph Search

- On small problems
  - Graph search almost always better than tree search
- On many real problems
  - Storage space is a huge concern.
  - Graph search impractical

گراف سرچ داره نودهای تکراری رو برامون ذخیره میکنه و از لحاظ اینکه ما به دور برنخوریم قابلیت اطمینانش این بیشتره ولی بحث مموری پیش میاد ینی تعداد زیادی نودها رو باید ذخیره بکنیم

Properties of A* Search

- Complete:
  - Yes, (if there is lower bound on cost)
- Time:
  - Exponential in (relative error in h x depth of Sol)
- Space:
  - $O(b^m)$—keeps all nodes in memory!(find best heuristic)
- Optimal:
  - Yes
- A∗ expands all nodes with f(n) < C∗, some nodes with f(n) = C∗, and no nodes with f(n) > C∗.

از لحاظ کامل بودن *A کامله
از لحاظ زمانی: متن رو بخون --> این هم به صورت نمایی است
از لحاظ فضا: b به توان m --> به صورت نمایی
و بهینه هم هست

نکته: الگوریتم *A اگر ما نودها رو تقسیم بندی بکنیم: یک تعدادی از این استیت ها توی مسیر بهینه هستند و یک تعدادی هم هیچ کاری به مسیر بهینه ندارند --> مسیر بهینه ما *C است حالا نودها رو می تونیم نسبت به این *C تقسیم بکنیم : یکسری نودها هستن که مقدار f اشون کمتر از *C است و *A حتما سراغ این ها می ره و یکسری نود هم داریم که مقدار f اشون برابر با خود *C است ینی توی مسیر رسیدن ما به هدف است و *A حتما اینارو بررسی میکنه و اونایی که مقدارشون بزرگتره ینی f بیشتر از *C است رو سراغشون هیچ وقت نمی ره چرا؟ چون ما همیشه f های کوچیک رو انتخاب میکنیم --> همین که این نودها رو ما داریم Expand می کنیم این یک چالش حافظه است چون ما اونایی که واقعا با خود *C مقدار f اشون برابره ما فقط دوست داریم اونا رو چک بکنیم و اینکه بیایم مسیرهای دیگه رو چک بکنیم و بفهمیم این مسیرها، مسیرهای بهینه نیستن این خودش یک بار حافظه داره سمت *A و ما دوست داریم این چالش حافظه رو کم بکنیم

But…

- A* keeps the entire explored region in memory
- => will run out of space before you get bored waiting for the answer
- There are variants that use less memory (Section 3.5.5):
  - IDA* works like iterative deepening, except it uses an *f*-limit instead of a depth limit
    - On each iteration, remember the smallest *f*-value that exceeds the current limit, use as new limit
    - Very inefficient when *f* is real-valued and each node has a unique value
  - SMA*
  - Beam Search
  - Recursive Best-first Search

چی کار میشه کرد؟
مشکل اینه که A* همه نودها رو که میخواد explored بکنه مجبوره توی حافظه نگه داره به امید اینه که اگر این نود رو انتخاب کرد و f کوچیکی داشت ممکنه وقتی که می ره جلو ببینیم اصلا این نود خوب نبوده و مجبوریم برگردیم و یک نود دیگه رو انتخاب بکنیم پس همه این نودها رو تا زمان رسیدن به هدف نگه داریم و این تعدادش می تونه خیلی زیاد باشه و توی خیلی از مسائل ما این حجم از حافظه رو نداریم
پس ما نیاز داریم این رو محدود بکنیم --> پس به کاری بکن در عین حال که از بهینه بودن خارج نمیشی این محدودیت حافظه هم در نظر بگیر
چندتا رویکرد است:
IDA*
SMA*
beam search
...

IDA*: Iterative Deepening A*

- To reduce the memory requirements at the expense of some additional computation time,

  Combine

  uninformed iterative deepening search with A*

- Use an *f-cost* limit instead of a depth limit

IDA*:
این روش مثل همون روش iterative deep search است که می گفتیم وقتی که میخوایم هدفی رو پیدا بکنیم توی روش جستجوی عمقی بیایم یک لیمیتی بذاریم ینی بیایم توی این درخت جستجو یکبار عمق 1 رو انالیز بکنیم به این امید که به هدف برسیم اگر رسیدیم که هیچی ولی اگر نرسیدیم عمق رو یکی بیشتر بکنیم ینی بریم درخت ها رو تا عمق 2 چک بکنیم به صورت عمقی و ویژگی که این داشت این بود که حافظه اش بسیار کم بود
حالا IDA* اومده از همین ایده کمک گرفته --> که همون رویه جستجوی عمقی رو داشته باشیم به صورت نااگاهانه ولی عمقمون رو از مقدار f value ها کمک بگیریم: ینی توی جستجوی عمقی می اومدیم عمق رو 1 یا 2 یا 3 یا 4 می گرفتیم که این همون اکشن هایی بود که انجام میدادیم ینی با یک اکشن به این برسیم یا دو اکشن که اینا عمق های ما میشد حالا اینجا ما روی اکشن ها هزینه داریم و یک heuristic
IDA* میاد حد سرچ عمقیش رو میداره روی f value ینی تا یک عمقی از f ها سرچ رو انجام میدیم

IDA*: Iterative Deepening A*

- In the first iteration, we determine a **"f-cost limit"**
  $f(n_0) = g(n_0) + h(n_0) = h(n_0)$, where n_0 is the start node.
- We expand nodes using the **depth-first algorithm** and backtrack whenever $f(n)$ for an expanded node n exceeds the cut-off value.
- If this search does not succeed, determine the **lowest f-value** among the nodes that were visited but not expanded.
- Use this f-value as the **new limit value** and do another depth-first search.
- Repeat this procedure until a goal node is found.

اینجا g(n0) میشه صفر
اینقدر این عملیات رو انجام میدیم تا به نود هدف برسیم ولی اگه نرسیدیم f رو افزایش بدیم
پیشنهاد برای افزایش f:
توی A* نودها براساس f value شون جلوی ما قرار دارن و اولین نودی که ما دیگه سراغش نمی ریم نودی خواهد بود که f value اش بزرگتر از اون حد لیمیت است و مبنایی که IDA* می ذاره اینه که توی گام بعدی مقدار f اون رو می ذاره به عنوان یکی لیمیت جدید و دوباره IDA* اجرا میشه و این کار رو اینقدر تکرار میکنیم تا به نود هدف برسیم

IDA* Algorithm - Top level

function IDA*(*problem*) **returns** *solution*

  root := Make-Node(Initial-State[*problem*])

  f-limit := f-Cost(*root*)

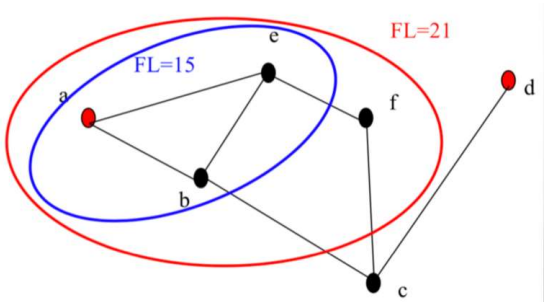
loop do

  solution, f-limit := DFS-Contour*(root, f-limit)*

  if *solution* is not null, **then return** *solution*

  if *f-limit = infinity,* **then return** *failure*

end



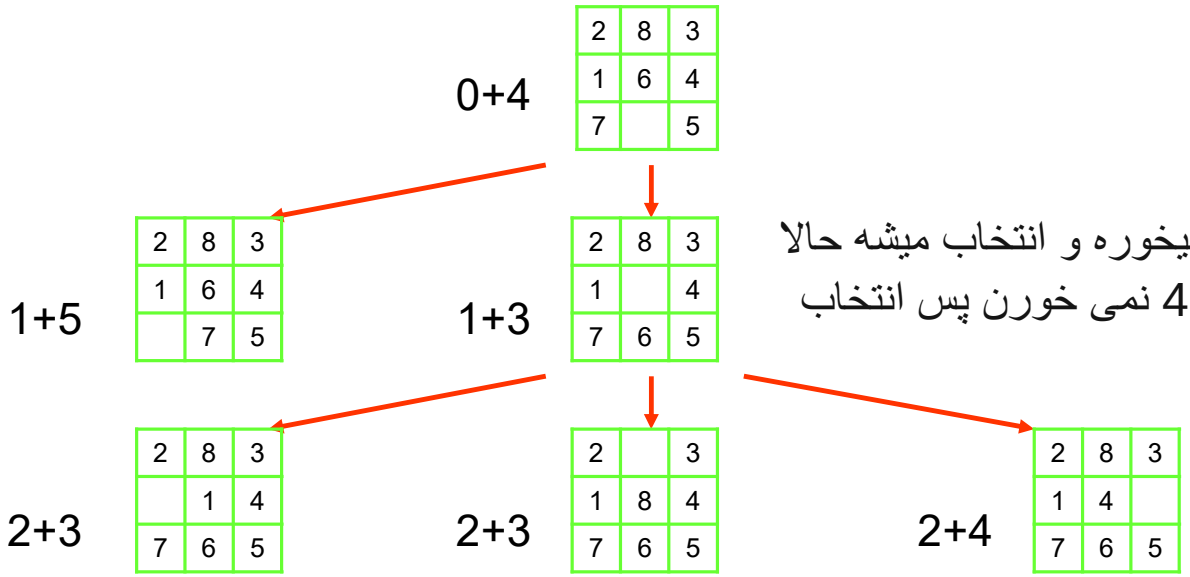
[40-417, SUT]

اینجا ولی هزینه سرچ می ره بالا

IDA* contour expansion

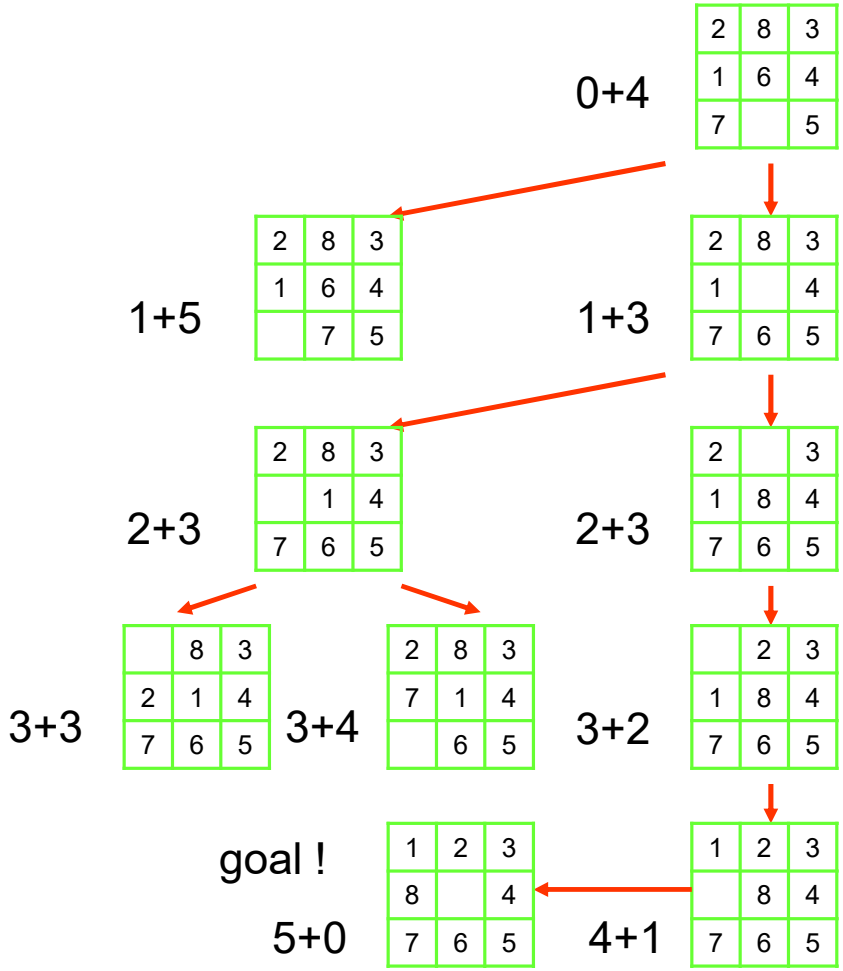
```
function DFS-Countour(node,f-limit) returns
        solution  sequence and new  f-limit
if f-Cost[node] > f-limit then return (null, f-Cost[node] )
if Goal-Test[problem](State[node]) then return (node,f-limit)
for each node s in Successor(node) do
   solution, new-f := DFS-Contour(s, f-limit)
  if solution is not null, then return (solution, f-limit)
   next-f := Min(next-f, new-f); end
return (null, next-f)
```

IDA* Example: Iteration 1, Limit = 4



این 4 میشه پس توی لیمیت میخوره و انتخاب میشه حالا بچه هاش هیچکدوم به لیمیت 4 نمی خورن پس انتخاب نمیشن

IDA* Example: Iteration 2, Limit = 5



[40-417, SUT]

Properties of IDA* Search

- Complete & Optimal (like A*)
  - Space usage ∝ depth of solution
  - Each iteration is DFS - no priority queue!
- nodes expanded relative to A* ?
  - Depends on # unique values of heuristic function
    - each f value is unique
    - In each step one new node add for A* expands

      $\Longrightarrow$ 1+2+...+n = O(n2)

      where n = nodes A* expands
    - if n is too big for main memory, n2 is too long to wait!
  - In 8 puzzle: few values $\Longrightarrow$close to # A* expands

نکته: ما اینجا دیگه یک صف اولویت نداریم --> صرفا یک نود رو داریم و بچه هاش رو و داریم عمقی باهاش برخورد میکنیم

رویه ای که برای اپدیت کردن داریم اینه که همیشه داره کوچکترین f که بعد از لیمیت ما است رو میذاره برای دور بعدی --> این یک چالشی ایجاد میکنه: مثلا ما 100 حالت داریم و 100 تا f مختلف ینی هیچ دو حالتی نیستن که fهاشون مثل هم باشه توی این حالت *IDA مثل این میشه که اول یه دونه و بعد دوتا و ... ینی هر دفعه انگار داره یک نود اضافه میشه و این اتفاق خوبی نیست چون این دنباله خودش توان 2 میکنه حافظه رو و اردر زمانی هم زیاد میشه
ولی اگر اینطوری نباشه ینی مقدار F ها تکراری باشه برای *IDA این خوبه --> بهترین حالتش اینه این *IDA توی حالت های گسسته خیلی خوبه

نکته: اگر لیمیت به جای مینیمم باشه ماکزیمم چه بلایی سر این الگوریتم میاد؟ این بهینه بودن رو تحت تاثیر قرار میده ینی به هدف می رسیم چون داره عمقی می ره ولی ممکنه از مسیر بهینه نرسیم

Simplified memory-bounded A*

- SMA* uses *all available memory* for the queue, minimizing thrashing
  - When full, drop worst node on the queue but remember its value in the parent

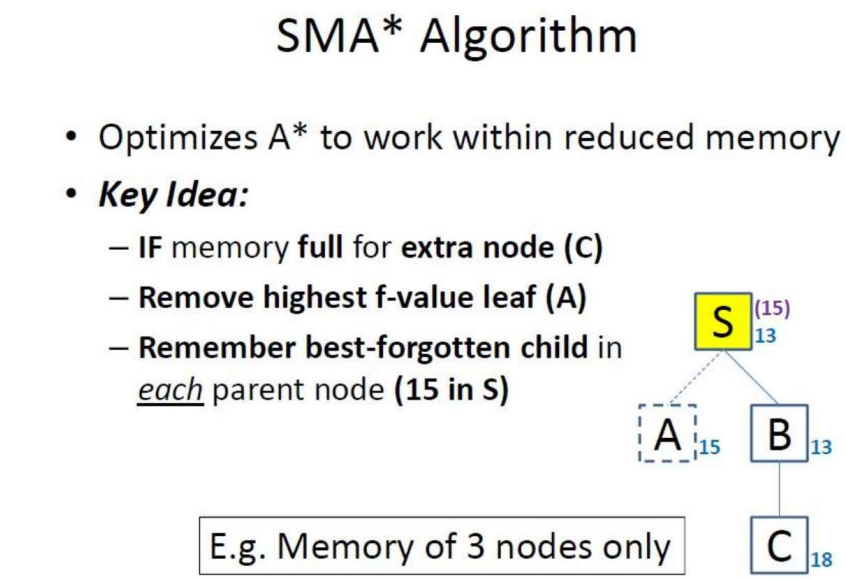
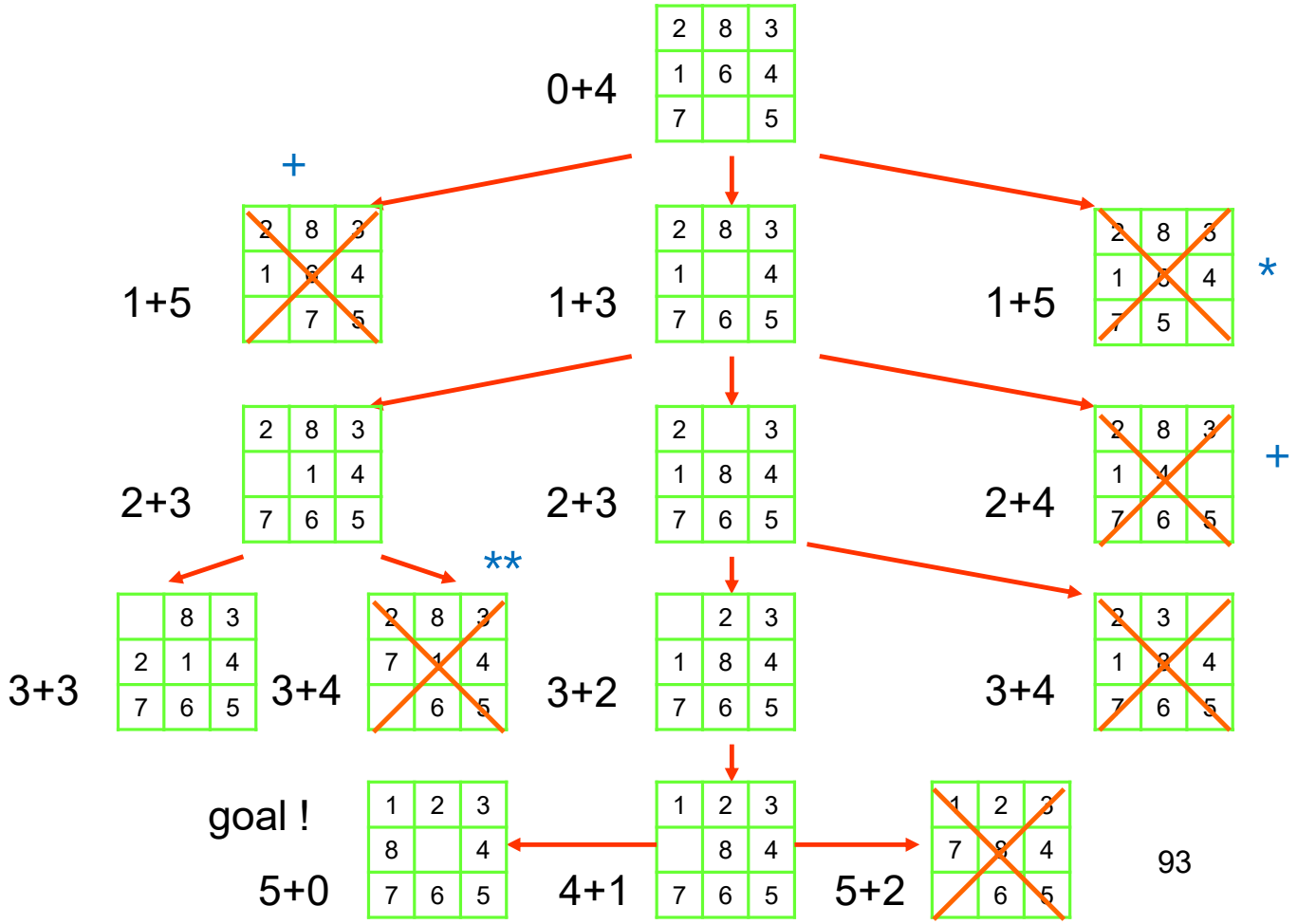


Image 1: Idea of how SMA* works

https://laconicml.com/simplified-memory-bounded-a-star/

:SMA*
ما توی رویه ای که میخوایم توی این A* سرچ بکنیم به چه چالشی برمیخوریم:
یه جایی صفمون اینقدر بزرگ میشه که ما دیگه نمی تونیم نود جدیدی بهش اضافه بکنیم
این روش میگه برای صف یک حدی بذار و اگر از یک حدی صف بزرگتر بود ینی نیاز داشتیم یک نود دیگه ای به صف اضافه کنیم میایم یکسری نود رو حذف بکنیم و فقط یک ردی از اونا رو نگه دار برای خودت همونایی که حذف کردی منظورمه مثلا بیا اطلاعات پدر اون نودی که حذف کردی رو نگه دار که اگر نیاز شد برگردی بدونی کدوم بوده که اینسرتش بکنی

Beam Search (with limit = 2)



[40-417, SUT]

Beam Search:
اینجا محدودیت حافظه داریم و نمی تونیم هر سه تاشو نگه داریم پس یکیشون رو میندازیم دور*
مرحله بعد دوتا از 6 ها رو حذف میکنیم ینی +
مرحله بعد 7 رو حذف میکنیم **
و همینطور می ره جلو...

و ممکنه این روش به جواب بهینه نرسه

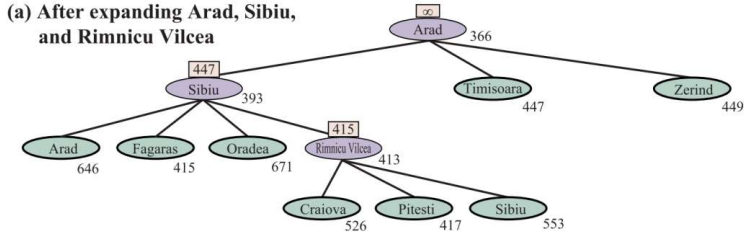
Recursive Best-first Search

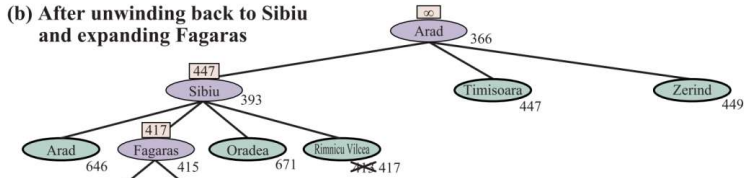
Best First search

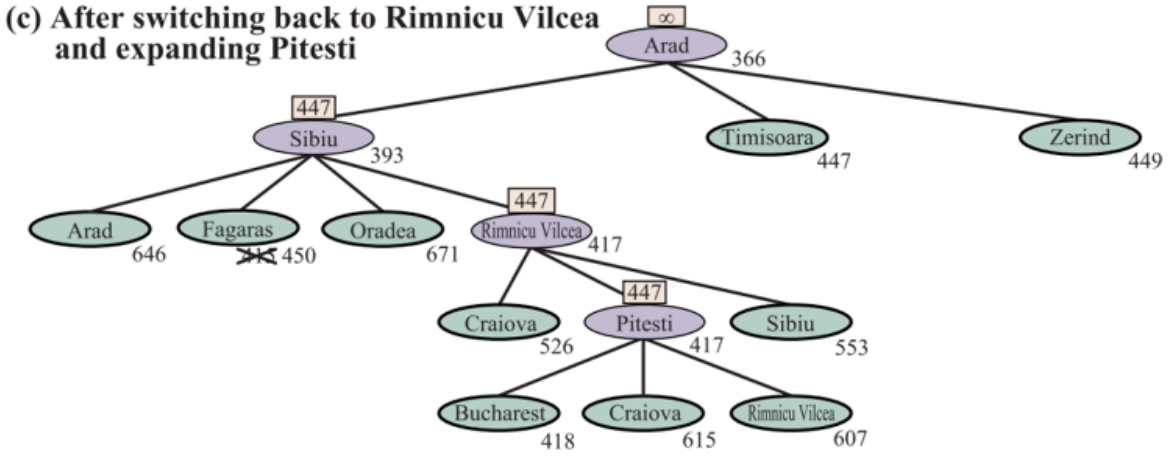
keep track of the f-value of the best alternative path available from any ancestor of the current n

Memory O(bd)

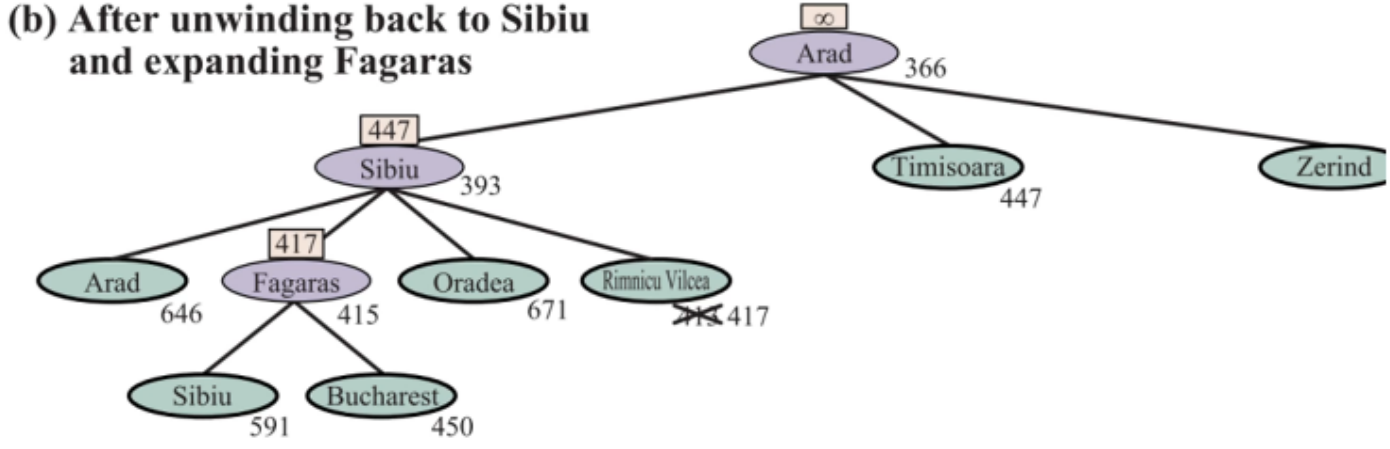
حافظه خیلی کم شد ینی به صورت خطی شده انگار







:Recursive Best-first Search
بهترین رو اول انتخاب بکن برای سرچ
بهترین چیه؟ براساس یک تابعی است مثلا می تونه عمق باشه
توی جستجوی اول بهترین یا اول بهترین رو انتخاب بکن براساس مقدار f اشون میایم جستجو رو انجام میدیم ینی اونی که f اش از همه بهتر رو انتخاب میکنیم و میخوایم حافظه هم محدود بکنیم ینی میخوایم همه نودها رو نگه داریم برای expand کردن میایم نود بهترین رو نگه می داریم و نود ما قبل بهترین
ینی سمت نود بهتر هی میخوایم عمیق بشیم
خب الان توی این مثال ما از arad می ریم به sibiu و f قبلی این هم ذخیره می کنیم ینی f مال timisoara که اگر از sibiu جواب نگرفتیم بریم سراغ اون یکی
الان بعد از sibiu می ره سراغ اونی که 415 است و فرزندان این یکیش 591 و یکیش 450 که جفت این ها از 415 بیشترن پس این کنسل میشه و می ریم سراغ اونی که 417 است و برای 415 هم میذاره 450 ینی بهترین فرزندش رو
برای بقیشون هم همینطور...



Pattern Database

- Underestimate is not good(manhattan in 8 puzzle)
- store these exact solution costs for every possible Pattern database subproblem instance(Pre-compute h* Heuristic based on Actual cost)
- Generate record
  - Use BFS for Sub problems
  - Create init-state from goal state

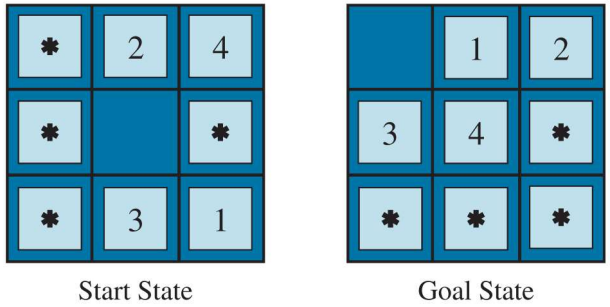


Figure 3.27 A subproblem of the 8-puzzle instance given in Figure 3.25. The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.

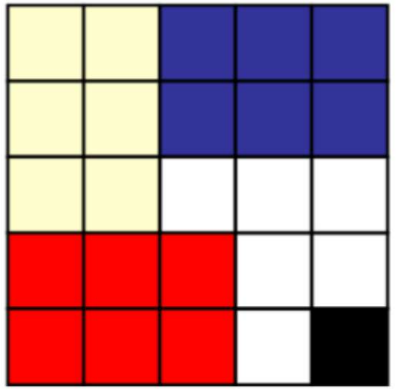
when solving random 15-puzzles can be reduced by a factor of 1000

:Pattern Database
یک پیشنهاد که وجود داره توی بحث A* اینه که ما هی میخوایم سرچ بکنیم و هی میخوایم Heuristic حساب بکنیم و این Heuristic ها حساب کردنش اگر بخوایم خیلی دقیق انجام بدیم زمان بر می شه و رفتیم سراغ این که این Heuristic رو ساده در نظر بگیریم
ما یک h* که همه Heuristic ها از این کمتر است و ایده ال این است که ما بریم سراغ Heuristic هایی که تا اونجایی که میشه به Heuristic ایده ال نزدیک تر بشه ولی بخاطر یه سری چیزا که سخت بود حساب کردنش و اینا سراغ این نمی ریم --> یک پیشنهاد اینه که از قبل بشینیم این ایده ال ها رو حساب بکنیم برای حالت های کوچیک ینی مسئله رو به یک تعدادی زیر مسئله تبدیل بکنیم و برای اون زیرمسئله ها یک Heuristic خیلی خوب حساب بکنیم و اینارو توی یک دیتابیسی ذخیره بکنیم
حالا شروع میکنه به سرچ کردن و به محض اینکه به یک زیر مسئله ای رسید که قبلا حساب کرده بود از دیتابیس می ره اون Heuristic زیر مسئله رو می خونه و با یک Heuristic خوب اون مسئله رو حل می کنه
مثال:
این مسئله 8 پازل است که یک حالت شروع داریم و به یک حالت هدف می خوایم برسیم و یک زیر مسئله ازاین مسئله 8 پازله چی می تونه باشه؟ ما دنبال این نباشیم که همه این ها رو مرتب بکنیم بلکه صرفا یک تعداد کاشی در نظر بگیریم و بگیم اگر اون تعداد کاشی رو بخوایم مرتب بکنیم چه مقدار هزینه داره و بقیه رو اسکیپ می کنیم و.... بعد که اینارو حساب کردیم توی یک دیتابیس ذخیره میکنیم و بعد مسئله 8 پازل رو با همه اعدادش حل میکنیم

این کار باعث میشه تهش Heuristic ما دقیق بشه
سرعت اجرا رو به شدت افزایش میده

Pattern Database

- Draw Back of Several Pattern Database
- Combine -> max(h1,h2)
  - Diminish return
  - Disjoin pattern data based
-



وقتی که ماکزیمم میگیریم همه عددها مثل هم میشه ینی یک تعدادی از Heuristic ها نزدیک بهم میشن و تقریبا ماکزیمم گیری خیلی کمک بهمون نمی کنه پس خیلی دنبال این هستیم که اینارو با هم جمع بکنیم
مثلا اگر یک Heuristic داشته باشیم که بین 1 تا 10 است و Heuristic دوم بین 1 تا 12 است در نهایت با ماکزیمم یک Heuristic پیدا میکنیم که بین 1 تا 12 است ینی 12 تاجایگاه برای سرچ داریم ولی اگر اینارو با هم جمع بکنیم و اجازش رو داشته باشیم دامنه ای که بهمون میده 1 تا 22 است ینی رزولویشن سرچ میره بالاتر
ایده پشت این:
میاد Heuristic فانکشن ها رو جوری تعریف میکنن که جمع پذیر بشن

نکته: جمع توی Heuristic ها اصلا کار درستی نیست و با یک شرایطی می تونیم جمع رو انجام بدیم

Alternative Approach

- Optimality is nice to have, but...
- Sometimes space is too vast!
  - Find suboptimal solution using local search.
