Chapter 3, Section 3.5

A* Search

Tracing, Properties, and Optimality

COMP338 - Artificial Intelligence

Birzeit University | Second Semester 2025/2026

Reference: Russell & Norvig, AIMA 4th Edition (2021), Section 3.5.2

Recap: Where Greedy Left Us

Last session: greedy best-first found a path but not the cheapest one.

Greedy result

Path: Arad - Sibiu - Fagaras - Bucharest
Cost: 140 + 99 + 211 = **450 km**

Why it failed: greedy only looks at h(n). It picked Fagaras because Fagaras LOOKED close (h=176). It ignored the path cost already paid.

True optimum

Path: Arad - Sibiu - Rim. V. - Pitesti - Bucharest
Cost: 140 + 80 + 97 + 101 = **418 km**

32 km cheaper. Today's algorithm finds this automatically.

Today's Lecture - Outline

1. **From Greedy to A***
   Define f(n) = g(n) + h(n). Why this single change matters.
2. **A* in Action: Romania**
   Step by step, every node expansion with full f-values and the priority queue shown.
3. **Properties of A***
   Complete, optimal, time, space. Compare to greedy and UCS.
4. **Admissibility**
   The precondition: h never overestimates the true cost.
5. **Optimality Proof**
   Why an admissible h guarantees that A finds the cheapest path.*
6. **Wrap-up**
   Summary and bridge to next session on memory-bounded search.

01

From Greedy to A*

Combine cost so far with estimated cost to come.

The Idea: Avoid Expensive Paths

Hart, Nilsson, and Raphael (1968)

Avoid expanding paths that are already expensive. Combine actual cost so far with estimated cost to come.

$$f(n) = g(n) + h(n)$$

estimated cost of the cheapest solution that goes through node n

g(n)

cost so far

Actual sum of step costs from the start to n. KNOWN.

h(n)

estimated remaining

Heuristic estimate of cost from n to the goal. GUESSED.

f(n)

estimated total

Sum of the two. Estimated total cost of a solution through n.

A* Generalises Greedy and UCS

A is a strict generalisation. Both algorithms we know are special cases.*

`f(n) = g(n)`

Set h(n) = 0 for every n

Uniform-Cost Search

Pure cost-so-far. Optimal but blind. Expands many nodes in directions away from the goal.

`f(n) = h(n)`

Set g(n) = 0 for every n

Greedy Best-First

Pure heuristic. Fast but neither complete nor optimal. Can be fooled by misleading h.

`f(n) = g(n) + h(n)`

Both g and h contribute

A* Search

Balanced. Complete and optimal (under admissibility). The right combination.

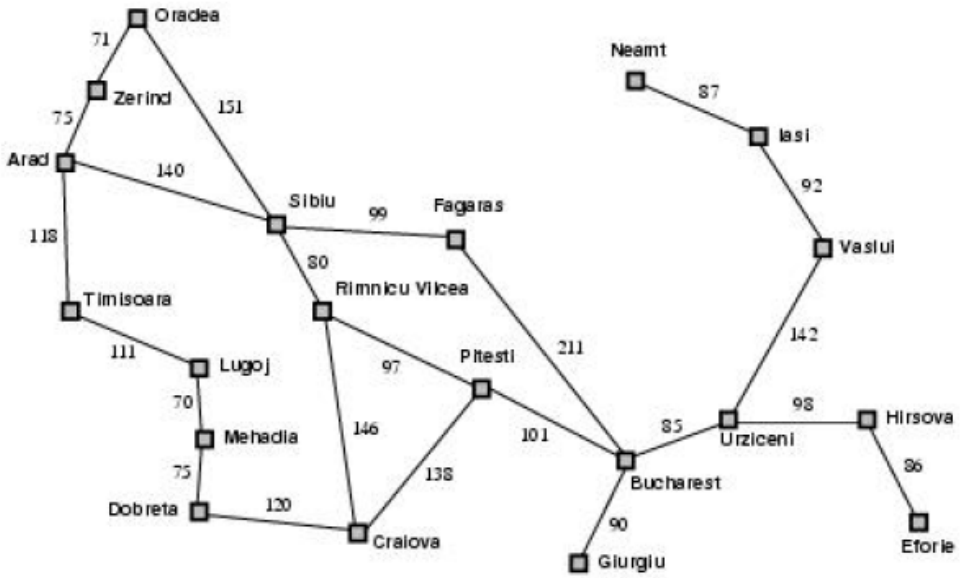
02

A* in Action: Romania

Same map, same heuristic, different result. Six steps.

The Romania Problem - Map and Data

Goal: shortest path from Arad to Bucharest. We use the AIMA map and the straight-line heuristic h_SLD.



Map of Romania. The numbers connecting cities represent distance in kilometers.

h_SLD - straight-line km to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176 Rim. V.: 193
Pitesti: 100 Craiova: 160
```

Edge costs (km, both directions)

```
Arad – Sibiu: 140
Arad – Timisoara: 118
Arad – Zerind: 75
Sibiu – Oradea: 151
Sibiu – Fagaras: 99
Sibiu – Rim. V.: 80
Fagaras – Bucharest: 211
Rim. V. – Pitesti: 97
Rim. V. – Craiova: 146
Pitesti – Bucharest: 101
Pitesti – Craiova: 138
```

Step 0 - Initial State

Start with only Arad on the priority queue. g(Arad) = 0. f(Arad) = 0 + 366 = 366.

Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140   A-T:118   A-Z:75
S-F:99    S-R:80    S-O:151
R-P:97    R-C:146   F-B:211
P-B:101   P-C:138
```

Priority Queue (sorted by f)

front -> *[smallest f popped first]*

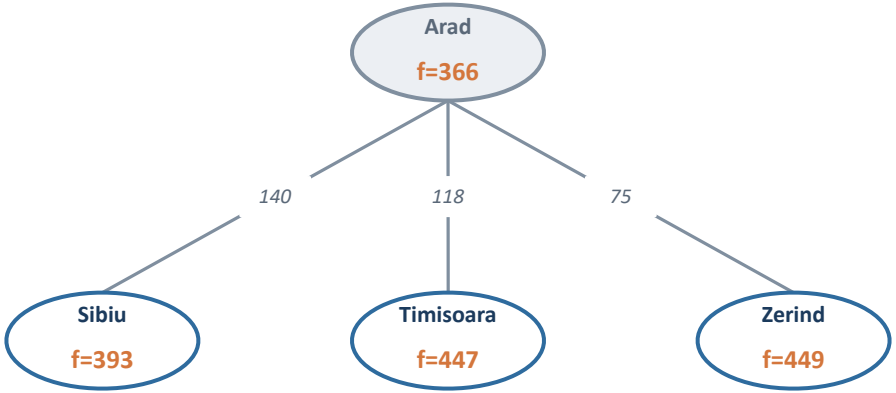
Arad	f = 366

Next action: pop the node with smallest f. Only one option.

A will pop Arad next, test if it is the goal, and (if not) expand it.*

Step 1 - Pop and Expand Arad

Pop Arad (f=366). Not the goal. Expand. Children: Sibiu, Timisoara, Zerind.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140  A-T:118  A-Z:75
S-F:99   S-R:80   S-O:151
R-P:97   R-C:146  F-B:211
P-B:101  P-C:138
```

Compute f = g + h for each child

```
Sibiu:     g = 0 + 140 = 140,  f = 140 + 253 = 393
Timisoara: g = 0 + 118 = 118,  f = 118 + 329 = 447
Zerind:    g = 0 + 75 = 75,    f = 75 + 374 = 449
```

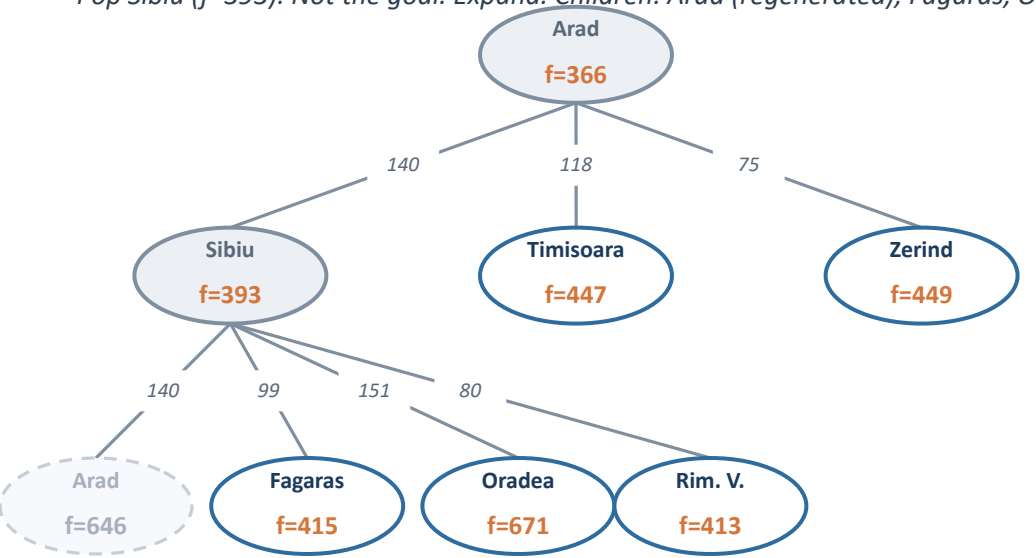
Priority Queue (sorted by f)

front -> *[smallest f popped first]*

Sibiu	**f = 393**
Timisoara	f = 447
Zerind	f = 449

Step 2 - Pop and Expand Sibiu

Pop Sibiu (f=393). Not the goal. Expand. Children: Arad (regenerated), Fagaras, Oradea, Rim. V.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140   A-T:118   A-Z:75
S-F:99    S-R:80    S-O:151
R-P:97    R-C:146   F-B:211
P-B:101   P-C:138
```

Priority Queue (sorted by f)

front -> [smallest f popped first]

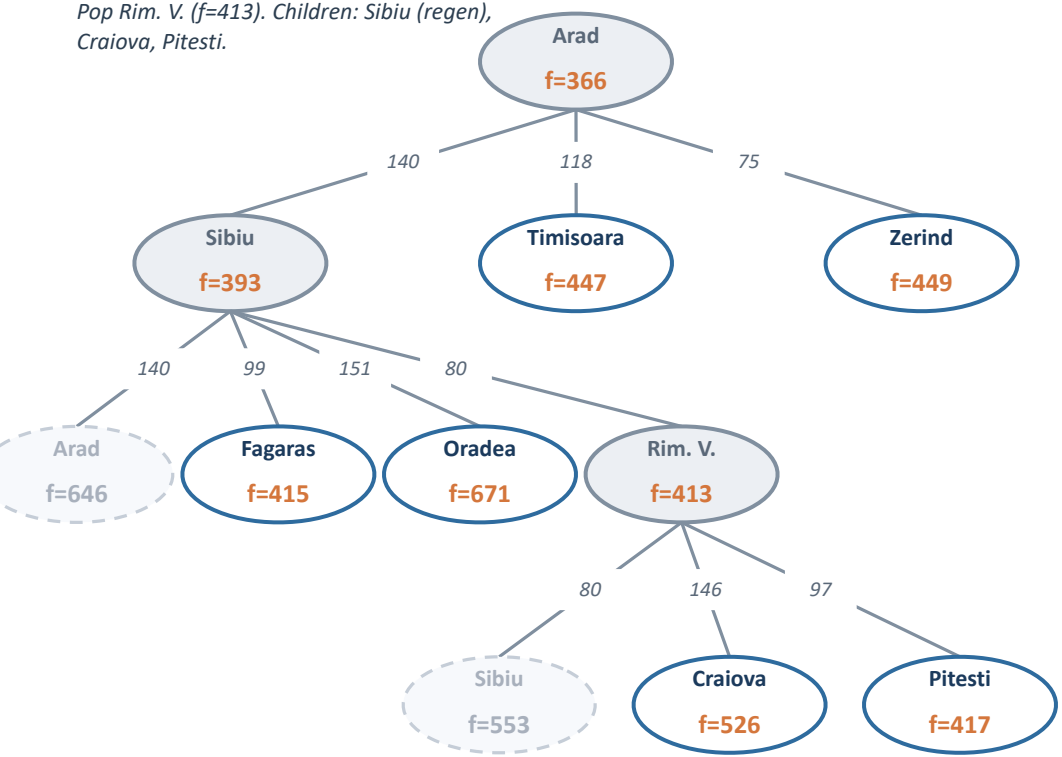
Rim. V.	**f = 413**
Fagaras	f = 415
Timisoara	f = 447
Zerind	f = 449
Oradea	f = 671
Arad (regen)	f = 646

This is the divergence from greedy

Greedy would pick Fagaras (h=176 smallest). A picks Rim. V. (f=413 < Fagaras f=415). The 19 km extra in h is more than offset by 19 km less in g.*

Step 3 - Pop and Expand Rimnicu Vilcea

Pop Rim. V. (f=413). Children: Sibiu (regen), Craiova, Pitesti.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140   A-T:118   A-Z:75
S-F:99    S-R:80    S-O:151
R-P:97    R-C:146   F-B:211
P-B:101   P-C:138
```

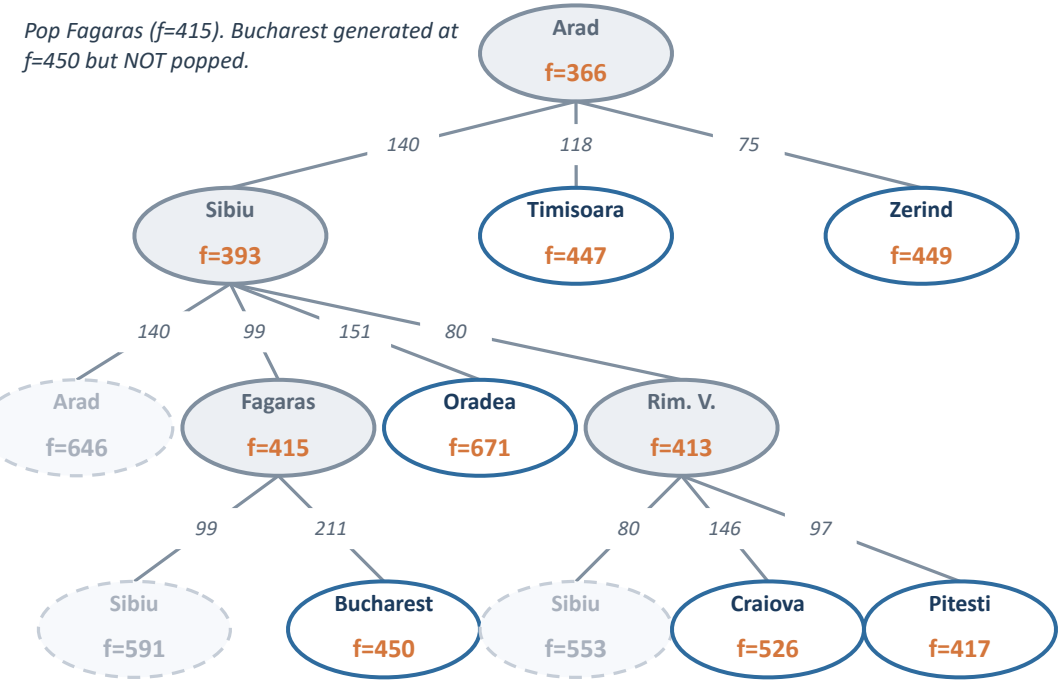
Priority Queue (sorted by f)

front -> *[smallest f popped first]*

Fagaras	**f = 415**
Pitesti	f = 417
Timisoara	f = 447
Zerind	f = 449
Craiova	f = 526
...	f = ...

Step 4 - Pop and Expand Fagaras

Pop Fagaras (f=415). Bucharest generated at f=450 but NOT popped.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140   A-T:118   A-Z:75
S-F:99    S-R:80    S-O:151
R-P:97    R-C:146   F-B:211
P-B:101   P-C:138
```

Priority Queue (sorted by f)

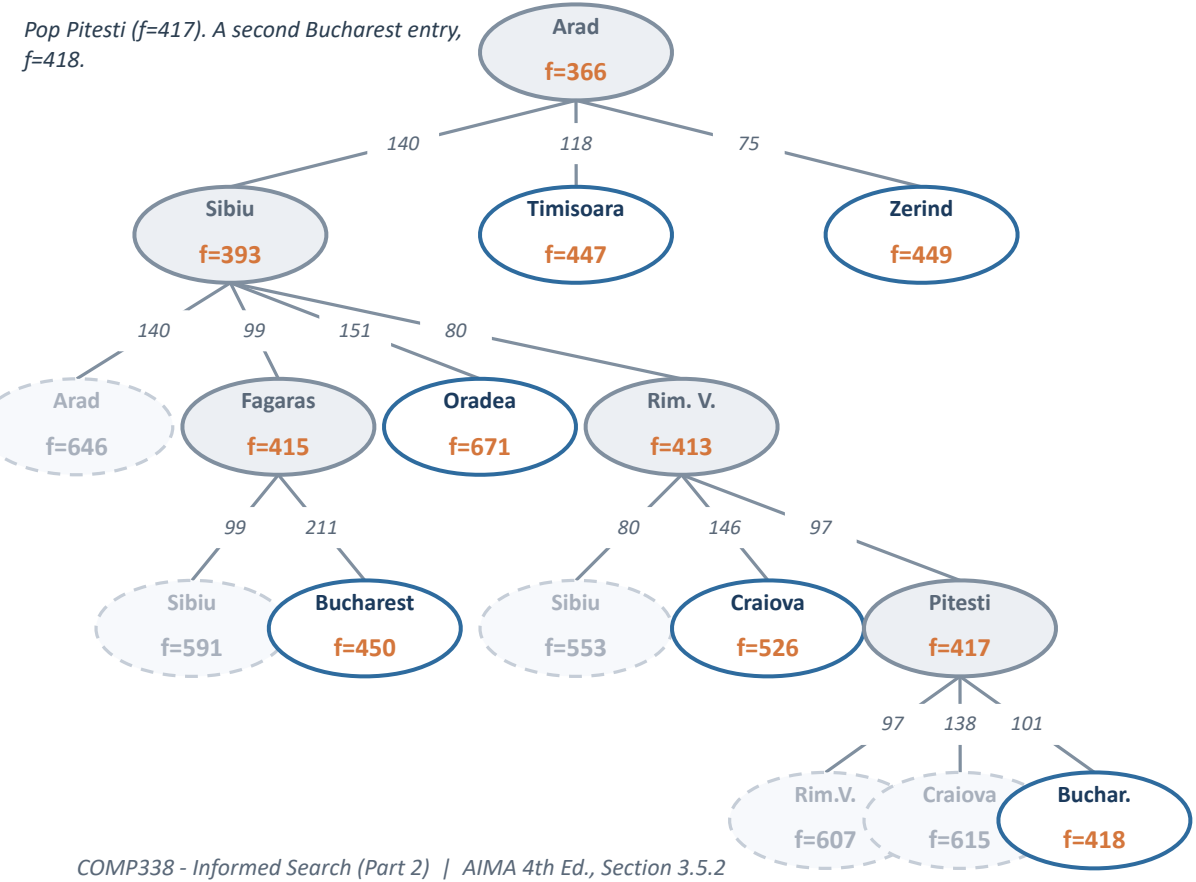
front -> *[smallest f popped first]*

Pitesti	**f = 417**
Timisoara	f = 447
Bucharest	**f = 450**
Zerind	f = 449
Craiova	f = 526
...	f = ...

Bucharest is in the queue at f=450, but Pitesti (f=417) is smaller. A* pops Pitesti next, not Bucharest.

Step 5 - Pop and Expand Pitesti

Pop Pitesti (f=417). A second Bucharest entry, f=418.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140  A-T:118  A-Z:75
S-F:99   S-R:80   S-O:151
R-P:97   R-C:146  F-B:211
P-B:101  P-C:138
```

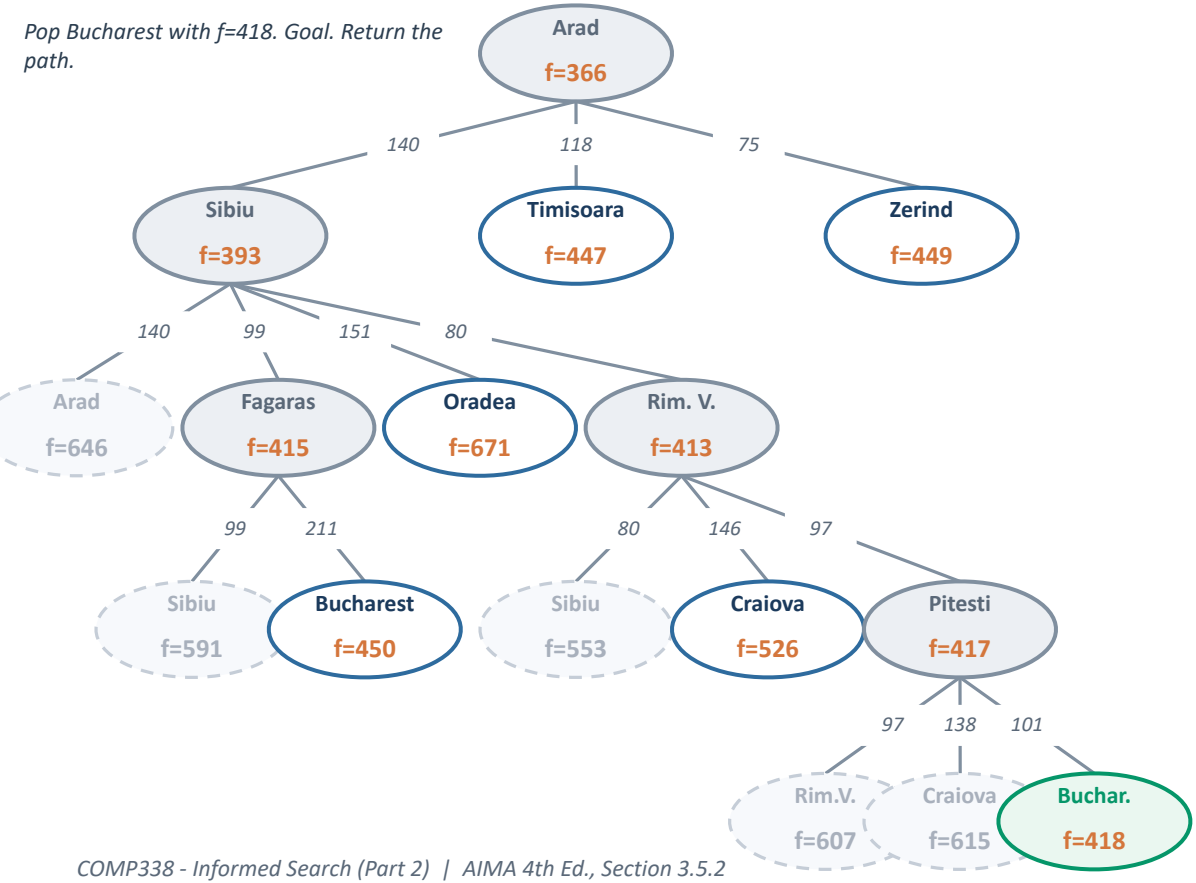
Priority Queue (sorted by f)

front -> *[smallest f popped first]*

Bucharest (Pitesti)	**f = 418**
Timisoara	f = 447
Zerind	f = 449
Bucharest (Fagaras)	f = 450
Craiova	f = 526
...	f = ...

Step 6 - Pop Bucharest - Done

Pop Bucharest with f=418. Goal. Return the path.



Solution found

Path:
Arad - Sibiu - Rim. V. - Pitesti - Bucharest

Cost: 140 + 80 + 97 + 101 = **418 km**
Optimal (we will prove this).

Search statistics (tree-search)

Distinct states expanded: 5
(Arad, Sibiu, Rim. V., Fagaras, Pitesti)

Regenerated nodes: 5
(redundant in tree-search; fixed by graph-search)

Greedy: 450 km. A*: 418 km.

32 km saved by including g(n).

03

Properties of A*

Complete, optimal, exponential time and space.

Properties of A*

Complete	Yes - if there are finitely many nodes with f <= f(G)
Optimal	Yes - if h is **admissible** (tree search) or **consistent** (graph search)
Time	Exponential in worst case. With a perfect h: linear. The better h, the more pruning.
Space	Exponential. Keeps every generated node in memory. THE practical limitation.

Memory is the catch.

A stores every generated node. On hard problems, it runs out of RAM before finding a solution. Next session: how to bound memory (IDA*, RBFS, SMA*).*

Comparing the Three Algorithms

Same problem (Romania, Arad to Bucharest). Three different evaluation functions.

Algorithm	f(n)	Romania result	Complete?	Optimal?
UCS	**g(n)**	418 km (correct)	**Yes**	**Yes**
Greedy	**h(n)**	450 km (wrong)	Graph: Yes	**No**
A*	**g(n) + h(n)**	**418 km (correct)**	**Yes**	**Yes (admissible h)**

UCS is also optimal. Why prefer A*?

Both find 418 km. UCS expands many more nodes because it has no sense of direction. A with a good heuristic prunes the irrelevant parts of the map. On a large map (millions of nodes), the difference can be 100x or more in expansions.*

04

Admissibility

The precondition for optimality. h never overestimates.

Admissibility

Definition (AIMA 4th Ed., Section 3.5.2)

A heuristic h is ***admissible*** if, for every node n,

h(n) <= h*(n)

where h(n) is the true cost of the cheapest path from n to a goal.*

In other words: an admissible h is OPTIMISTIC. It never claims the goal is farther than it really is.

h_SLD on Romania

Admissible. Straight-line distance is always less than or equal to road distance. Roads are not straight.

h(n) = 0

Trivially admissible. But useless - A with h=0 is just Uniform-Cost Search. Loses all guidance.*

h(n) = 1.1 * h_SLD(n)

Inflated. Can overestimate. NOT admissible. A with this h is no longer guaranteed optimal.*

05

Optimality Proof

Why admissible h guarantees A returns the cheapest path.*

Optimality Theorem for A*

Theorem (AIMA Section 3.5.2)

If h is **admissible**, then A* using **TREE-SEARCH** is **optimal**.
That is, the first goal A pops from the queue lies on a cheapest path.*

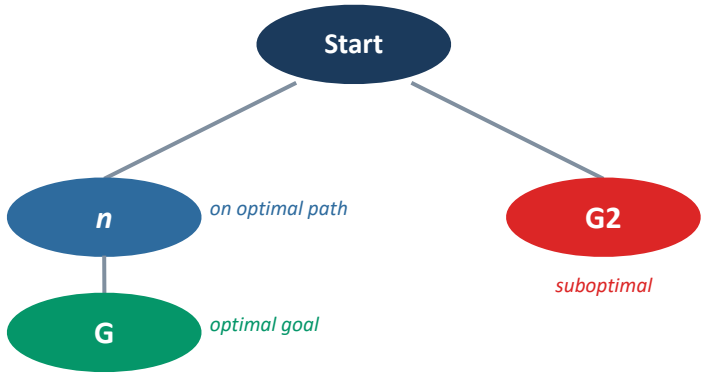
Proof setup

Definitions

C* = cost of an optimal path. G = an optimal goal (cost = C*).
G2 = some suboptimal goal (cost > C*). n = an unexpanded node on the path Start to G.

Proof strategy (by contradiction)

Suppose A* pops G2 first. Show: f(n) < f(G2).
Then A would have popped n before G2 - a contradiction.*



Optimality Proof - Three Inequalities

1. $f(G2) = g(G2) + h(G2) = g(G2)$

h(G2) = 0 because G2 is a goal

2. $f(G2) = g(G2) > C^*$

*G2 is suboptimal, so its cost exceeds the optimal cost C**

3. $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^*$

h(n) <= h(n) by admissibility; g(n) + h*(n) = C* because n is on the optimal path*

Combining (2) and (3): f(n) <= C* < f(G2)

So f(n) < f(G2). A always pops smallest f first, so A* would have popped n before G2. Contradiction. QED.*

06

Wrap-up

Summary and what is next.

Summary

f(n) = g(n) + h(n)	A* combines actual cost so far with estimated cost to come.
Romania: 418 km, optimal	A* found Arad - Sibiu - Rim. V. - Pitesti - Bucharest. 32 km better than greedy.
Goal test at POP, not GENERATE	A* must wait until the goal is popped. Bucharest sat in the queue at f=450 while Pitesti at f=417 was expanded first.
Tree-search regenerates parents	Without a reached set, A* re-explores states (e.g., Sibiu under Rim. V.). Graph-search adds a reached set to skip them.
Admissibility: h(n) <= h*(n)	The precondition. Optimistic heuristic never overestimates the true cost.
Optimality: proven	Three inequalities show A* never expands a suboptimal goal before the optimal one.

References and Further Reading

Primary textbook

Russell, S. & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th Ed.). Pearson.
Chapter 3, Section 3.5.2 (A* search). Figure 3.18 shows the Romania trace.

Original paper and supplementary notes

Hart, P., Nilsson, N. & Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Trans. Systems Science and Cybernetics 4(2), 100-107.
Jarrar, M. (2018). *Heuristic Informed Search Algorithms*. Birzeit University.
http://www.jarrar.info/courses/AI/Jarrar.LectureNotes.Ch3.InformedSearch.pdf

Online resources

Drexel CS - A* Romania Greatest Hits (suggested map source): cs.drexel.edu/~popyack/Courses/AI/Fa20/interludes/Interlude_A*_Romania_GreatestHits.html
Amit Patel, Introduction to A* (interactive): https://www.redblobgames.com/pathfinding/a-star/introduction.html
UC Berkeley CS188 lectures on A*: https://inst.eecs.berkeley.edu/~cs188/

Chapter 3, Section 3.5.5

Memory-Bounded Heuristic Search

IDA, RBFS, and SMA**

COMP338 - Artificial Intelligence

Birzeit University | Second Semester 2025/2026

Reference: Russell & Norvig, AIMA 4th Edition (2021), Section 3.5.5

Recap: A* Has a Memory Problem

A is optimal, but it stores every generated node. On hard problems, memory is the limit.*

What A* gives us

- Complete (finite state spaces)
- Optimal under admissibility
- Smart pruning via f = g + h
- Fewer expansions than UCS in practice

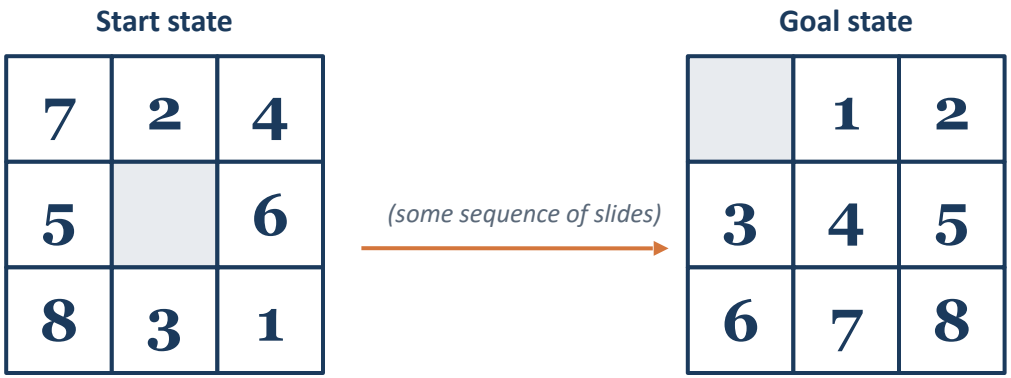
What A* takes from us

- Frontier grows exponentially in solution depth
- Reached set also grows exponentially
- RAM exhausts before solution found
- *Often a much bigger limit than time*

AIMA: "memory is usually a worse problem for A than time"*

Concrete Example: The 8-Puzzle

Before we look at numbers, here is a problem you can hold in your hand.



How the 8-puzzle works

States: any arrangement of 8 tiles + blank on a 3x3 grid.

Actions: slide a tile into the adjacent blank.

Branching factor: ~3 (blank has 2-4 neighbours).

Reachable states: 9! / 2 = 181,440

(only half - parity invariant)

Average optimal depth: **~22 moves**

(hardest cases reach 31)

Keep this picture in mind. On the next slide we scale up - same rules, bigger grid - and watch A fail.*

How Bad Is the Memory Problem?

Same rules, bigger grids. The numbers explode.

Problem	Reachable states	Typical depth	A* memory needed	Verdict
8-puzzle (3x3)	~1.8 x 10^5	~22	thousands	**A* OK**
15-puzzle (4x4)	~1.0 x 10^13	~50	billions	**A* fails**
24-puzzle (5x5)	~7.8 x 10^24	~100	intractable	**A* fails**
Continental routing	depends on map	varies	GB of state	A* often fails

The Korf (1985) breakthrough

Richard Korf solved the 15-puzzle - which had defeated A on the same hardware - using a new algorithm called IDA*. Same optimality guarantee as A*, but linear memory.*

What Is A* Actually Storing?

Each node A generates becomes a record in memory. Records live in two structures.*

Per-node record (~48 bytes)

state - the configuration itself (e.g. the 3x3 grid)
parent pointer - to reconstruct the path on success
g(n) - actual cost from start to this node
h(n) - heuristic estimate to the goal
f(n) = g + h - priority queue ordering key

Two structures hold these records

Frontier (priority queue): nodes generated but not yet expanded. Sorted by f. A* pops the smallest.

Reached set: states already expanded. Prevents re-expansion on revisit.

Why A* cannot throw any of this away

Drop the frontier? A* may pop D next, then discover D's children are all expensive. The next-best could be a node already in the frontier (say E with f=17). Drop E and that path is gone forever.
Drop the reached set? A different path may re-generate the same state. Without the set, A* re-expands it - children, grandchildren, exponential blow-up from revisits.
Drop parent pointers? When the goal is finally popped, the only way to return the actual path is to walk parents back to the start. No pointers = no path.

Two Strategies to Bound Memory

All three algorithms today use one of these two ideas.

Strategy A: Iterate

- Don't store alternatives. Store only the current path.
- When you fail, restart with a slightly larger limit.
- Cost: re-expand the same nodes across iterations.
- **Benefit: linear memory.**

Used by: IDA, RBFS.*

Strategy B: Use all available memory

- Run A* normally until memory fills up.
- When full, drop the WORST node (highest f).
- Remember the dropped value at the parent so we can reconsider it later.
- Cost: still exponential time. May not be optimal if memory is too small.
- **Benefit: best use of available RAM.**

Used by: SMA.*

Today's Lecture - Outline

1. **IDA* - Iterative Deepening A***

   Apply IDS to f-cost. Linear memory. Worked iteration trace.

2. **RBFS - Recursive Best-First Search**

   Recursive DFS with backed-up f-values. Three-stage Romania trace.

3. **SMA* - Memory-Bounded A***

   Use all available memory. Drop the worst node. Step-by-step example.

4. **Designing Good Heuristics**

   Admissibility, dominance, relaxed problems.

5. **Comparison and Wrap-up**

   Side-by-side properties of A, IDA*, RBFS, SMA*.*

01

IDA* - Iterative Deepening A*

Iterative deepening on f-cost. The breakthrough for the 15-puzzle.

IDA*: The Core Idea

Idea (Korf, 1985)

Recall IDS from Part 1: do DFS with depth limit 0, then 1, then 2... until you find a solution.
IDA* does the same, but with f-cost limit instead of depth limit.

1

Initial f-limit

Set f-limit = h(start). Run a DFS, pruning any branch where f(n) > f-limit. If a goal is found at or under the limit, return it.

2

Update the f-limit

If no goal found, set the new f-limit to the SMALLEST f-value that was pruned. This guarantees the next iteration sees new nodes.

3

Repeat

Keep iterating with progressively larger f-limits until a goal is found.

IDA*: Annotated Pseudocode

```
function IDA*(problem):
    f_limit  <-  h(start)
    loop:
        result, next_lim  <-  DFS(start, f_limit)
        if result is solution: return result
        if next_lim = infinity: return failure
        f_limit  <-  next_lim

function DFS(node, f_limit):
    if f(node) > f_limit:
        return failure, f(node)
    if GOAL(node):
        return SOLUTION, f(node)
    next_lim  <-  infinity
    for each child c of node:
        res, new_lim  <-  DFS(c, f_limit)
        if res is solution: return res
        next_lim <- min(next_lim, new_lim)
    return failure, next_lim
```

Initialise budget

f_limit starts at h(start) - smallest possible f any solution can have.

Iterate budgets

Each loop iteration runs one bounded DFS, then raises the budget.

Exit when no progress

next_lim = infinity means nothing was pruned and no goal found. No solution.

Prune over-budget

If f exceeds budget, abandon this branch. Return f as a candidate next-iteration limit.

Goal test after prune

Goal test happens AFTER the prune check, so we only commit to in-budget goals.

Recurse on children

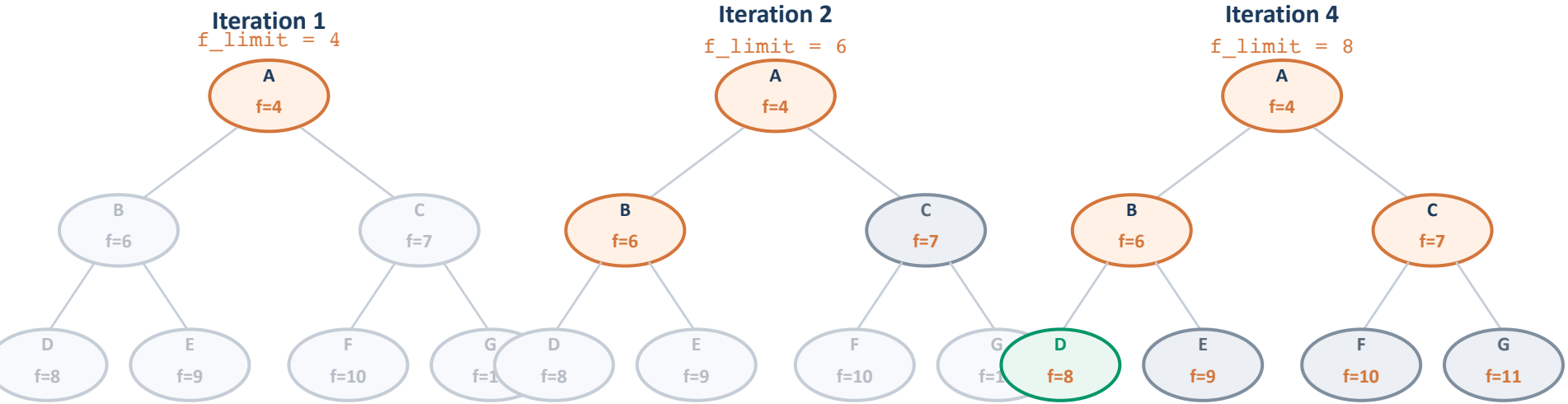
Standard DFS. Track the smallest pruned f across all children.

Bubble up the limit

On failure, return our smallest pruned f so the parent can aggregate.

IDA*: Worked Example - Iterations

Small abstract tree, costs labeled, h-values shown. Three iterations until a goal is found.



Visit A (f=4). Children all have f > 4. Prune. Next limit = 6 (smallest pruned).

Visit A, B (f=6). B's children have f > 6, prune. C (f=7 > 6), prune. Next limit = 7. (Iter. 3 at limit=7 expands C similarly, next limit = 8.)

Visit A, B, then D (f=8, goal). Return D. Optimal solution found.

IDA*: Properties

Complete

Yes, in finite state spaces with finite step costs.

If a solution exists at some cost C, the iteration with f_limit = C will find it - the goal is in budget that iteration.

Optimal

Yes, when h is admissible.

The first iteration to find a goal has f_limit = C. Any cheaper goal would have been found in an earlier iteration. Strict prune check (f > limit) means f = C* is in budget.*

Time

Depends on number of distinct f-values. Excellent for unit-cost problems, poor for fine-grained costs.

Each iteration repeats earlier work. With few distinct f-values (puzzles), iterations are sparse and the LAST one dominates. With continuous costs, each iteration adds one node - quadratic total work.

Space

O(b * d) - linear in solution depth. The key win.

DFS only needs the call stack: one frame per node on the current path. No frontier list, no reached set. For 15-puzzle, just ~240 nodes vs A's billions.*

02

RBFS - Recursive Best-First Search

Recursive DFS with backed-up f-values.

RBFS: The Core Idea

Idea (Korf, 1993)

Mimic best-first search using only linear space, by being a recursive depth-first search that **keeps track of the best alternative f-value** from any ancestor.
When the current path's f exceeds this alternative, the recursion unwinds, and the backed-up f-value is stored at the ancestor.

Like DFS, but smart

- Follows the best path down recursively.
- Stores f-values of siblings on the call stack.
- When current f exceeds best sibling f, unwind.
- Replace the parent's f with the best child f (the backed-up value).

Re-expansion is allowed

If the backed-up f at a forgotten subtree becomes the new minimum, RBFS re-explores it.

This change of mind lets RBFS recover from heuristic mistakes while staying optimal.

RBFS: Annotated Pseudocode (AIMA Fig. 3.22)

```
function RBFS(problem, node, f_limit):
    if GOAL(node):
        return SOLUTION(node)
    children  <-  EXPAND(node)
    if children is empty:
        return failure, infinity
    for each c in children:
        c.f  <-  max(g(c)+h(c), node.f)
    loop:
        best  <-  child with smallest f
        if best.f > f_limit:
            return failure, best.f
        alt  <-  second-smallest f
        result, best.f  <-
            RBFS(problem, best,
                 min(f_limit, alt))
        if result != failure:
            return result
```

Goal check on entry
Test the moment we recurse into a node. Returning later would risk a suboptimal goal.

Generate children, handle dead end
Empty successors means no path through here. Return infinity so the parent abandons this branch.

Inherit parent's f
max(g+h, node.f) preserves backed-up values across calls. Without this, RBFS would forget what it learned.

Pick best, check budget
If even the best child is over budget, unwind. Return best.f as the backed-up value for the parent.

Get the alternative
The second-smallest f. Becomes part of the new recursion budget.

Recurse with tighter limit
Pass min(parent budget, alternative). Ensures we will switch if current path exceeds alternative.

Loop: re-pick best
After recursion returns updated f, best may have changed. Loop and re-pick.

RBFS on Romania: Setup

Same problem (Arad to Bucharest). Same heuristic h_SLD. Watch RBFS oscillate.

What to watch for

- Each call carries an f_limit (best alternative from any ancestor).
- When the best child's f exceeds f_limit, RBFS unwinds and backs up that f.
- Backed-up values shown in purple brackets, e.g. **[417]**.

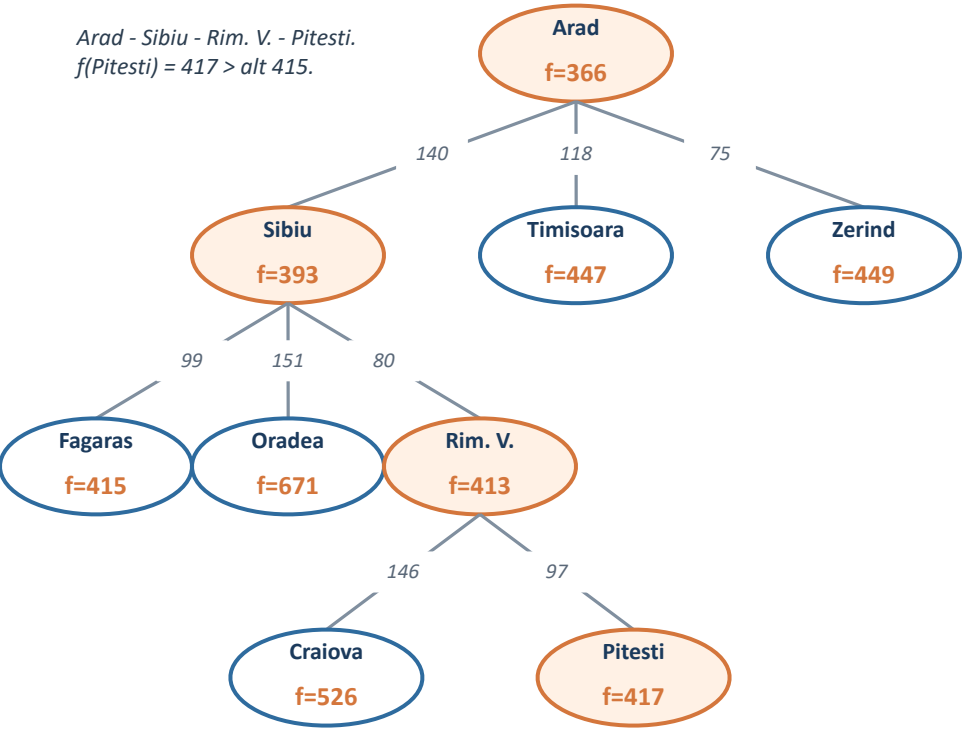
Reminder: f-values we will see (from Part 2)

```
f(Sibiu) = 393     f(Rim. V.) = 413     f(Fagaras) = 415     f(Pitesti) = 417
f(Bucharest via Fagaras) = 450     f(Bucharest via Pitesti) = 418

Three RBFS stages:
  (a) Follow path to Pitesti. f(Pitesti)=417 > alternative Fagaras=415. Unwind, back up 417 to Rim. V.
  (b) Try Fagaras. Reach Bucharest at f=450 > backed-up 417. Unwind, back up 450 to Fagaras.
  (c) Re-expand Rim. V. (now 417 < 450). Reach Bucharest at f=418. Solution.
```

RBFS Stage (a) - Down to Pitesti

Arad - Sibiu - Rim. V. - Pitesti.
f(Pitesti) = 417 > alt 415.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

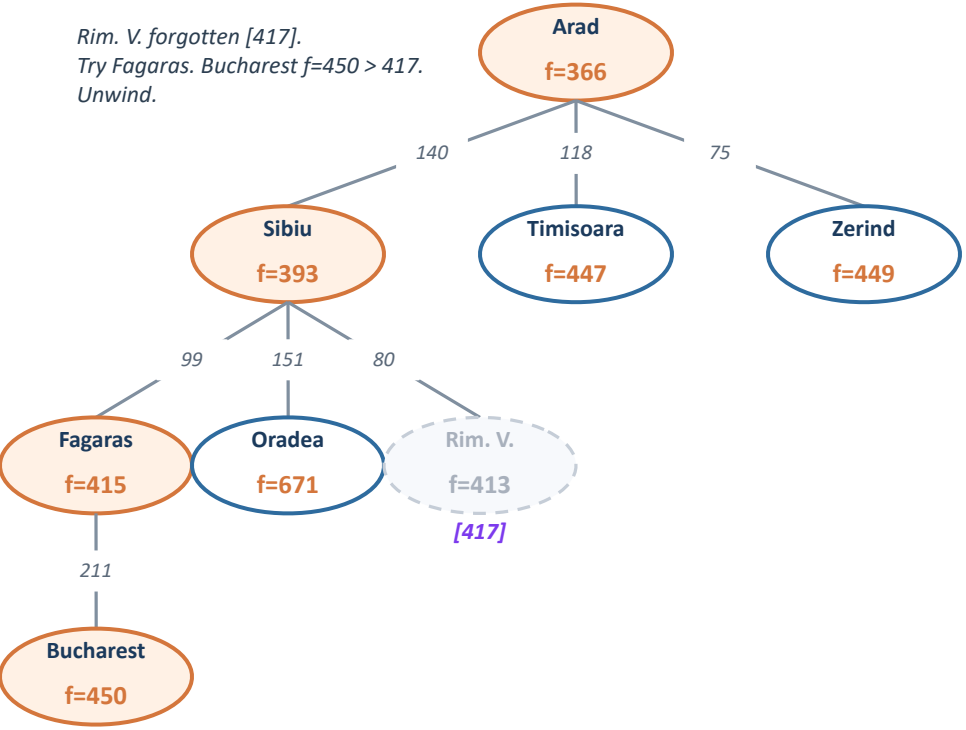
```
A-S:140   A-T:118   A-Z:75
S-F:99    S-R:80    S-O:151
R-P:97    R-C:146   F-B:211
P-B:101   P-C:138
```

Recursion state

Current node:	**Pitesti**
f(Pitesti):	**417**
f_limit:	**415**
Best alt above:	Fagaras (415)
Decision:	**417 > 415 -> unwind, back up 417**

RBFS Stage (b) - Try Fagaras Path

Rim. V. forgotten [417].
Try Fagaras. Bucharest f=450 > 417.
Unwind.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

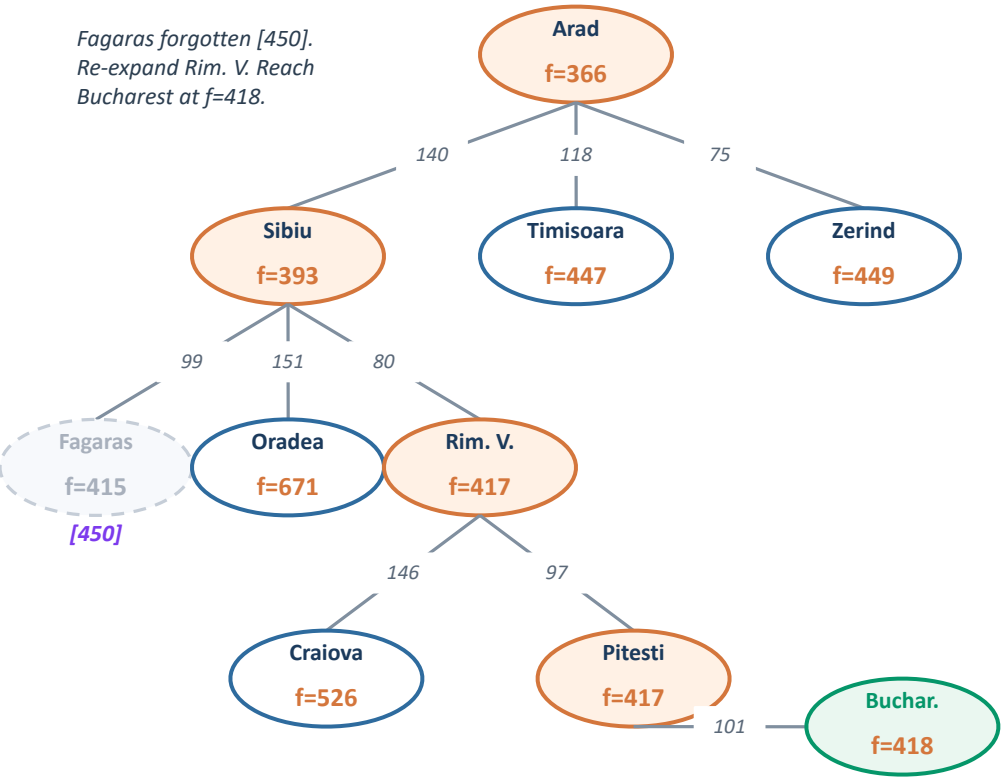
```
A-S:140  A-T:118  A-Z:75
S-F:99   S-R:80   S-O:151
R-P:97   R-C:146  F-B:211
P-B:101  P-C:138
```

Recursion state

Current node:	**Bucharest (via F)**
f(Bucharest):	**450**
f_limit:	**417**
Best alt above:	Rim. V. (now 417)
Decision:	**450 > 417 -> unwind, back up 450**

RBFS Stage (c) - Back to Rim. V., Solve

Fagaras forgotten [450].
Re-expand Rim. V. Reach Bucharest at f=418.



Reference: h(n) to Bucharest

```
Arad: 366     Bucharest: 0
Sibiu: 253    Timisoara: 329
Zerind: 374   Oradea: 380
Fagaras: 176  Rim. V.: 193
Pitesti: 100  Craiova: 160
```

Edge costs used

```
A-S:140   A-T:118   A-Z:75
S-F:99    S-R:80    S-O:151
R-P:97    R-C:146   F-B:211
P-B:101   P-C:138
```

Solution found

Path: Arad - Sibiu - Rim. V. - Pitesti - Bucharest
Cost: **418 km** *(same as A*)*

Memory used: only the current path. Linear.

RBFS: Properties

Complete

Yes, in finite state spaces.

Same argument as IDA: any solution at cost C is reachable when f_limit eventually grows to C.*

Optimal

Yes, when h is admissible.

Goal test only on entry to a node committed to as best path. Backed-up f-values preserve information across re-entries, so RBFS never settles for a suboptimal goal.

Time

Often faster than IDA*, especially with real-valued costs. Worst case exponential.

Backed-up values avoid redundant re-exploration. Where IDA restarts from the root each iteration, RBFS unwinds only to the level where the alternative becomes attractive.*

Space

O(b * d) - linear. Same order as IDA*.

Each call frame stores the children's f-values (b numbers). Stack depth bounded by solution depth d. Slightly larger constant factor than IDA due to backed-up bookkeeping.*

03

SMA* - Memory-Bounded A*

Use ALL available memory. Drop the worst node when full.

SMA*: The Core Idea

Idea (Russell, 1992)

Run A* normally. When memory fills up, **drop the worst leaf node** (highest f). Back its value up to its parent so the branch can be reconsidered later.

1 Expand best

Pick the leaf with smallest f. Expand it.

2 Memory full? Drop worst

Drop the leaf with highest f. Tie-break: oldest (shallowest) first.

3 Back up the value

Store the dropped f at the parent. Update ancestors as needed.

4 Max depth, not goal? f = infinity

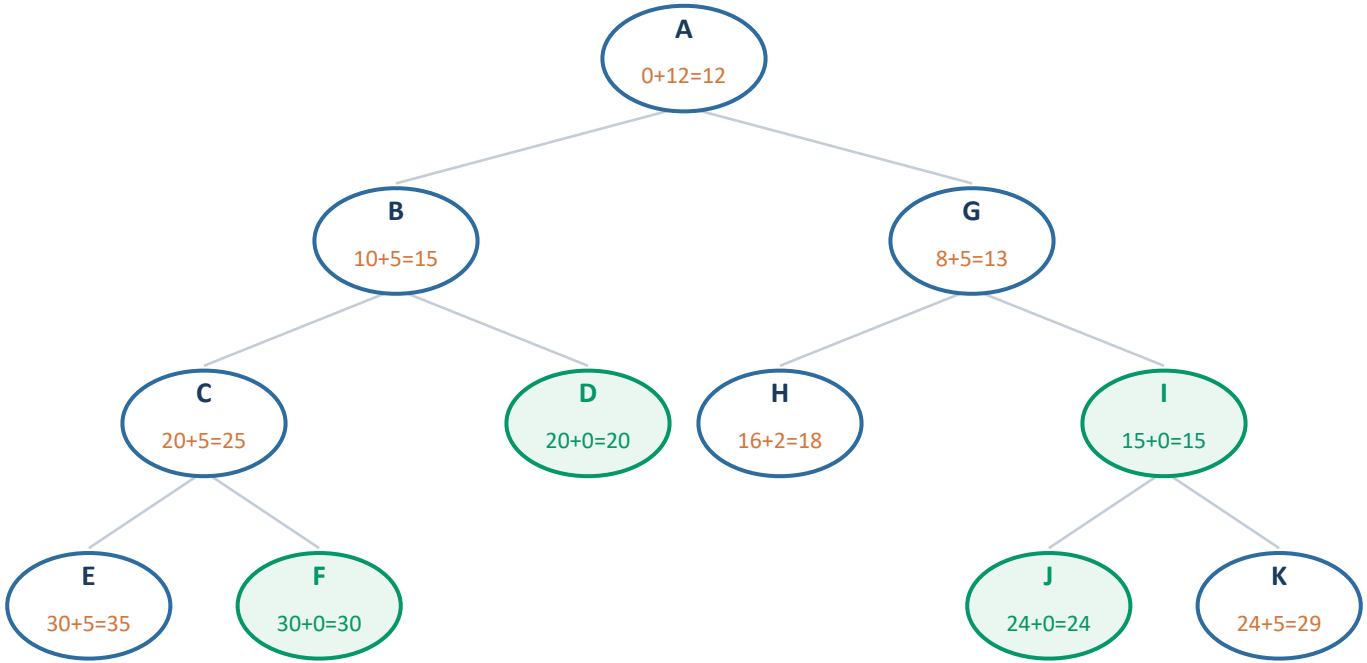
If a generated successor sits at the memory-depth boundary and is not a goal, set its f to infinity.

5 Re-expand when needed

If a forgotten branch's backed-up value is now smallest, re-expand it.

SMA* Example: Search Space

Small tree from AIMA/Jarrar. Each leaf labeled g + h = f. Goal nodes shaded green. Memory limit = 3 nodes.



Goals: D (f=20), I (f=15), F (f=30), J (f=24). Optimal goal: **I with cost 15.** *SMA* will find I despite the 3-node memory limit.*

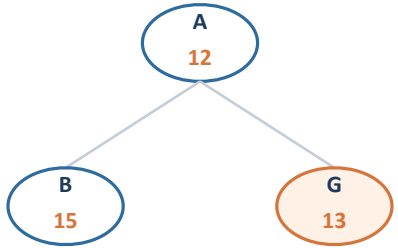
SMA* Trace - Steps 1 to 3: Expand A, then G

Three snapshots of memory. Each shows the tree fragment currently held. Memory limit = 3 nodes.

Step 1: Initialize

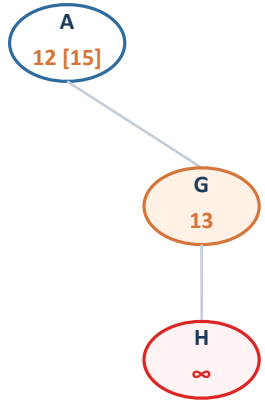
Memory = {A(12)}.
A is the start. Expand it next.

Step 2: Expand A



Generate children B(15), G(13).
Memory = {A, B, G}. Full.
Best leaf = G (smallest f).

Step 3: Expand G, generate H

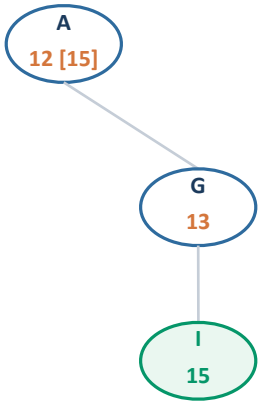


Expand G. H is at max depth, not goal: f(H) = infinity.
Drop B (next worst). Back up B's f=15 to A.

SMA* Trace - Steps 4 to 6: Find and Verify Goal I

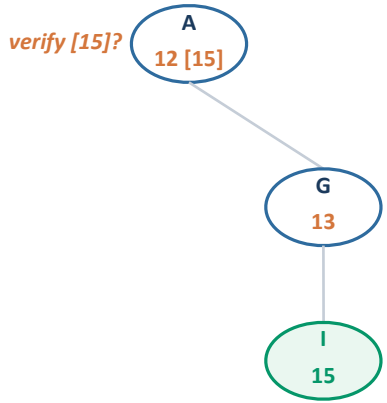
H is now a dead end. The next-best leaf to explore is I.

Step 4: Replace H with I



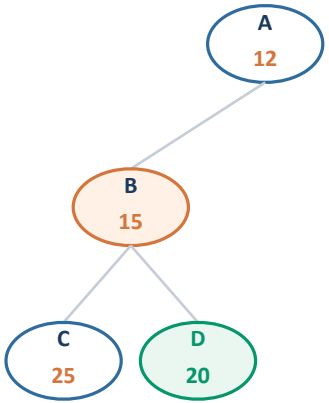
H is infinity - useless. Drop H. Generate G's other child I (f=15). I is a GOAL.
Memory = {A[15], G, I}.

Step 5: Solution candidate



I is a goal at f=15.
But A has [15] backed up from B. Could B's subtree have a cheaper goal?
Need to check.

Step 6: Re-expand B



Drop I and G (we will return if needed).
Re-expand B. Children C(25), D(20).
B's min child f = 20.

SMA* Trace - Final: Return Optimal Solution

*B's best child (f=20) is worse than I (f=15). So **I** at cost 15 is the optimal goal.*

Comparing candidate solutions

From G subtree:
I (goal) at f = 15

From B subtree:
C at f = 25 (not a goal)
D (goal) at f = 20

Best in B: 20 **Best in G: 15**

15 < 20, so I is the optimal goal.

Solution returned

Path: A → G → I

Cost: 15

Memory used: 3 nodes (the bound).

Re-expansions: B's subtree was re-explored once to verify optimality.

What made this work:

Backed-up f-values let SMA* know what was forgotten and re-investigate when needed.

SMA*: Properties

Complete

Yes, if available memory can hold the shallowest solution.

Memory must accommodate at least one full root-to-goal path simultaneously. Otherwise we cannot represent the solution even if we find it.

Optimal

Yes, if memory holds the shallowest optimal solution path. Otherwise returns best solution that fits.

SMA never returns a goal that is dominated by a known better one. But with too little memory, it may forget the optimal path entirely before completing the verification.*

Time

Worst case exponential. With sufficient memory, comparable to A*.

With memory >= A's frontier size, SMA* behaves identically to A*. With less, time degrades as re-expansions multiply. Bookkeeping overhead is significant.*

Space

Exactly the memory you allow. Uses all of it.

The defining feature: SMA takes a memory bound as a parameter and fills it. No wasted RAM, no surprise crashes from over-allocation.*

04

Designing Good Heuristics

Admissibility, dominance, and how to construct heuristics from relaxed problems.

Admissibility

Definition

A heuristic h(n) is **admissible** if for every node n, $h(n) \leq h^*(n)$, where h*(n) is the true optimal cost from n to a goal. An admissible heuristic **never overestimates**. It is optimistic.

8-puzzle sample state

5	4	
6	1	8
7	3	2

Start state S

Goal

1	2	3
4	5	6
7	8	

Two admissible heuristics for the 8-puzzle

`h1(n) =` number of misplaced tiles

for state S above: `h1(S) = 8` *(all 8 tiles are out of place)*

`h2(n) =` total Manhattan distance *(sum of each tile's distance from its goal position, ignoring collisions)*

for state S above: `h2(S) = 3+1+2+2+2+3+3+2 = 18`

---

*Both **never overestimate**: each tile must move at least Manhattan distance, so h2 is a tighter underestimate than h1.*

Dominance

Definition

If `h2(n) ≥ h1(n)` for all n, and both are admissible, then **h2 dominates h1**. The dominant heuristic expands fewer nodes.

Empirical: nodes expanded on 15-puzzle

Depth	IDS	A* with h1	A* with h2
d = 12	3,644,035	227	73
d = 24	too many	39,135	1,641

At depth 24, A with h2 expands ~24x fewer nodes than h1.*
Manhattan distance is a strict win on this benchmark.

Why dominance shrinks expansions

A* expands every node with f(n) < C*.

If h2 ≥ h1 everywhere, then f2 ≥ f1.

More nodes cross the f > C* threshold with h2 - so more are pruned.

Every node A(h2) expands is also expanded by A*(h1). The reverse is not true.*

Relaxed Problems

Theorem

A **relaxed problem** is the original problem with fewer restrictions on actions. The cost of an optimal solution to a relaxed problem is an **admissible heuristic** for the original problem.

Why admissible: *any solution to the original is also a solution to the relaxed (the relaxed has fewer constraints, accepts more solutions). So the relaxed optimum is at most the original optimum.*

Relaxation 1: tile can move anywhere

Original rule: tile moves to **adjacent blank square**.
Relaxed rule: tile teleports anywhere.

Optimal solution to relaxed problem: one move per misplaced tile.

Heuristic: `h1 = number of misplaced tiles`

Relaxation 2: tile can move to any adjacent square

Original rule: tile moves to **adjacent blank square**.
Relaxed rule: tile slides over occupied squares too.

Optimal solution to relaxed problem: each tile moves directly to its goal, distance = Manhattan.

Heuristic: `h2 = total Manhattan distance`

05

Comparison and Wrap-up

Side by side. When to use which algorithm.

Comparison of Heuristic Search Algorithms

Algorithm	Complete?	Optimal?	Time	Space
A*	Yes	Yes (admissible)	Exponential	**Exponential**
IDA*	Yes	Yes (admissible)	Exponential (re-expansion)	**O(b*d) linear**
RBFS	Yes	Yes (admissible)	Exponential, often less than IDA*	**O(b*d) linear**
SMA*	If memory enough	If memory enough	Exponential	**Bounded by user**

When to use which

Plenty of memory + small problem: A*. **Tiny memory + unit-cost puzzle:** IDA*.
Tiny memory + real-valued costs: RBFS. **Fixed memory budget + want to use it all:** SMA*.

Summary

Memory is A*'s weakness	Exponential frontier. Crashes on hard problems like 15-puzzle.
Two strategies: iterate, or use all memory	Strategy A (IDA*, RBFS): forget and re-try. Strategy B (SMA*): use every byte.
IDA* = IDS with f-cost limit	Set f-limit = h(start), DFS, prune f > limit, increase limit. Linear memory, possible re-expansion.
RBFS = recursive DFS with backed-up f	Remember best f in forgotten subtree. Re-expand when attractive. Less re-expansion than IDA*.
SMA* = A* + drop worst when full	Uses all memory. Optimal if memory permits. Complex bookkeeping.
All optimal under admissibility	Memory bounding does not break the theoretical guarantee. Just slows down or limits problem size.

References and Further Reading

Primary textbook

Russell, S. & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th Ed.). Pearson.
Section 3.5.5 (Memory-bounded search). Figure 3.22 (RBFS pseudocode), Figure 3.23 (Romania trace), Figure 3.25 (SMA* example).

Original papers

Korf, R. (1985). *Depth-first iterative-deepening: An optimal admissible tree search*. Artificial Intelligence 27(1), 97-109. (IDA*)
Korf, R. (1993). *Linear-space best-first search*. Artificial Intelligence 62(1), 41-78. (RBFS)
Russell, S. (1992). *Efficient memory-bounded search methods*. ECAI 1992. (SMA*)

Local course materials and online

Jarrar, M. (2018). *Heuristic Informed Search Algorithms*. Birzeit University. (Slides 33-40)
http://www.jarrar.info/courses/AI/Jarrar.LectureNotes.Ch3.InformedSearch.pdf
UC Berkeley CS188 (memory-bounded search): https://inst.eecs.berkeley.edu/~cs188/