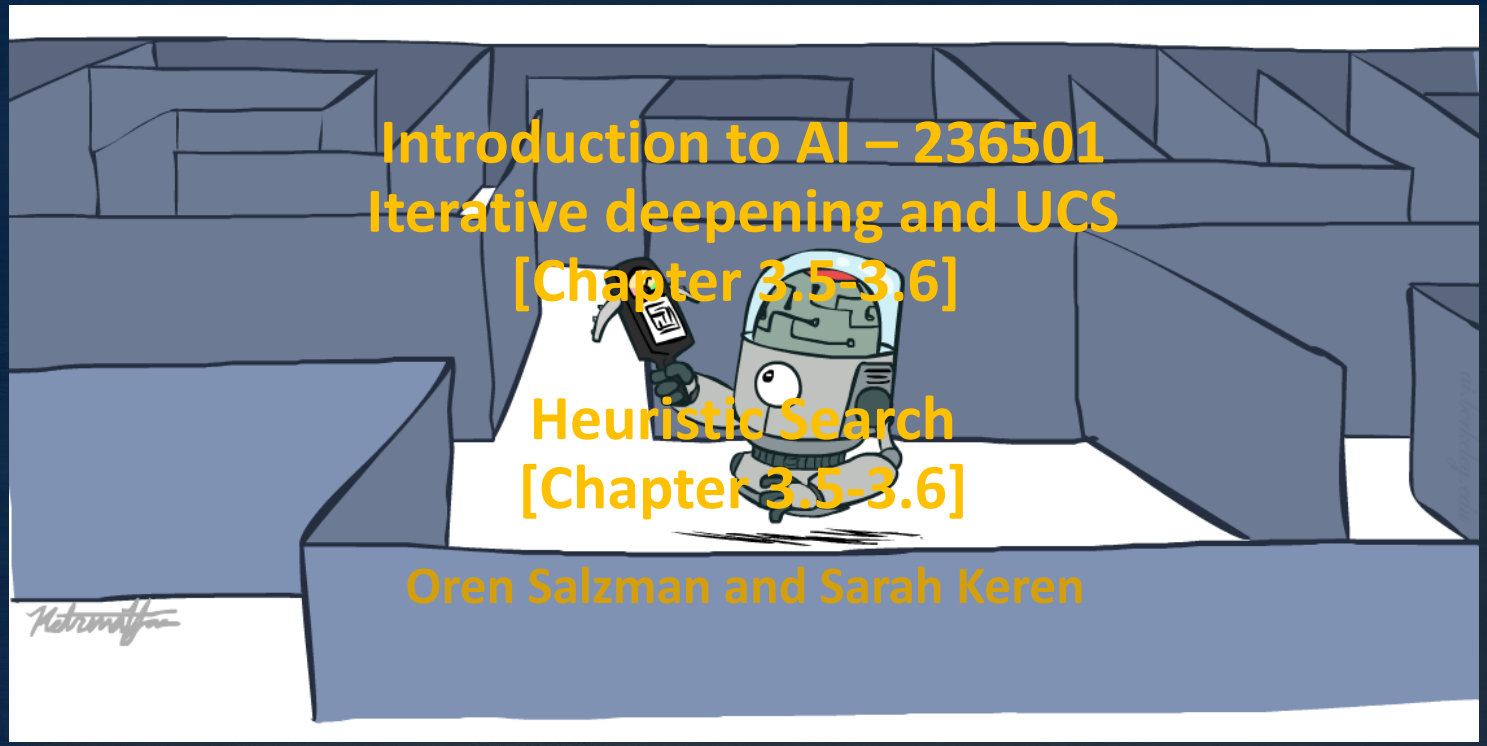


Introduction to AI – 236501
Iterative deepening and UCS
[Chapter 3.5-3.6]

Heuristic Search
[Chapter 3.5-3.6]

Oren Salzman and Sarah Keren

Slides adapted from ai.berkeley.edu by

Dan Klein, Pieter Abbeel and Anca Dragan

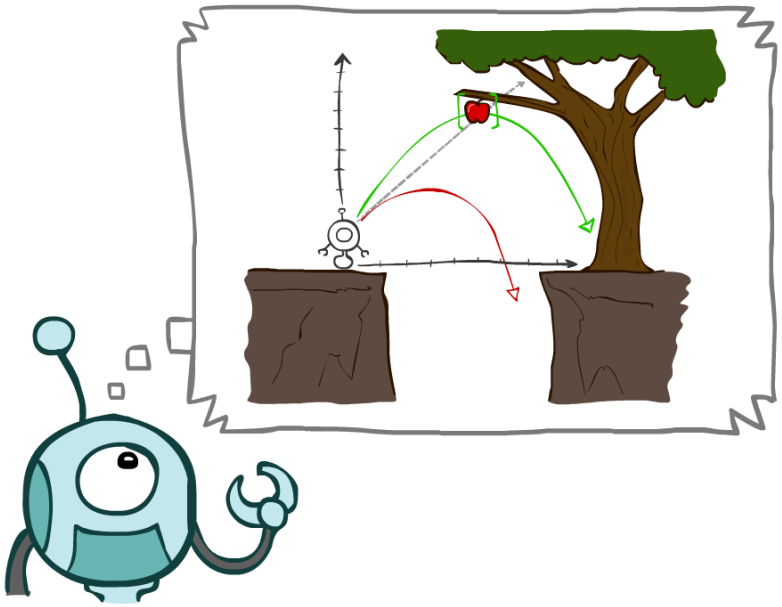
and from Shaul Markovitz @ Technion

TECHNION
Israel Institute of Technology

The Henry and Marilyn Taub
Faculty of Computer Science

Today

- Uninformed Search Methods
  - Iterative deepening
  - Uniform-Cost Search
- Informed Search methods
  - Greedy Best-First Search
  - A*
  - Heuristics
  - IDA*



General Tree Search (recap)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

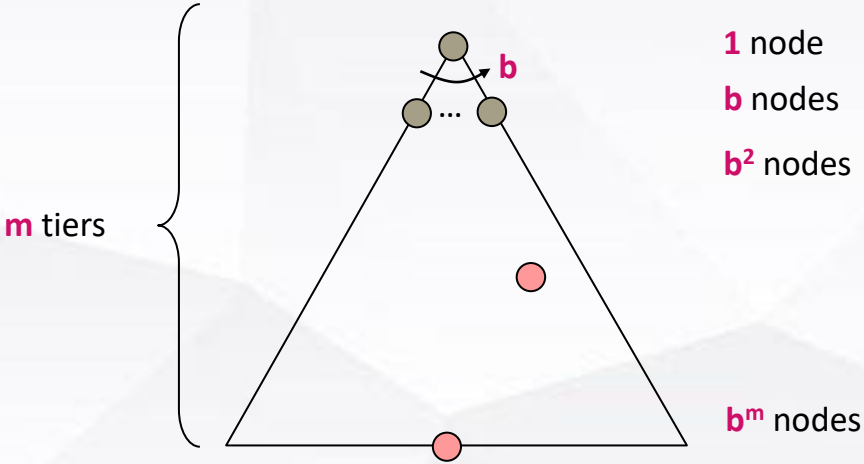
- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

Search Algorithm Properties

- **Complete**: Guaranteed to find a solution if one exists?
- **Optimal**: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - **b** is the branching factor
  - **m** is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots + b^m = O(b^m)$



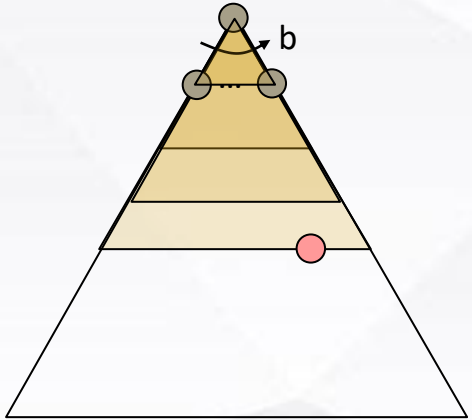
Iterative deepening

TECHNION
Israel Institute of Technology

The Henry and Marilyn Taub
Faculty of Computer Science

Iterative Deepening

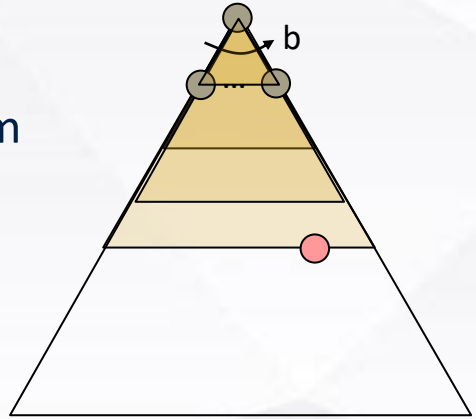
- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..



- **Isn't that wastefully redundant?**
  - Generally most work happens in the lowest level searched, so not so bad!

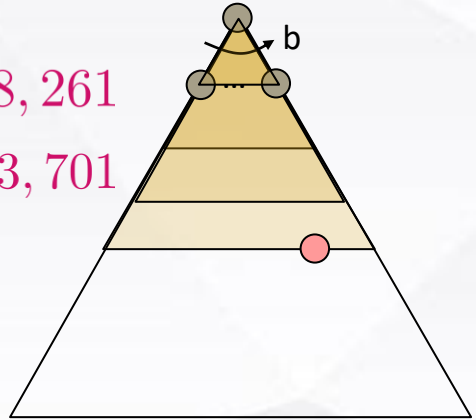
Iterative Deepening - properties

- **Completeness** – if a solution of length **L** exists, `ID` will find it in the **L**'th iteration
- **Optimality -** if `ID` found a solution in the **L**'th iteration, there cannot be a solution of length **<L** (assuming uniform cost edges)
- **Memory complexity** – linear in solution depth (similar to `DFS-L` which is linear in the bound **L**)
- **Time complexity** –
  - If solution has length **L**, the search tree will have **L+1** layers
  - If the branching factor is b the **i**'th layer will have b^i nodes
  - $t = \sum_{i=0}^{L} \underbrace{(L+1-i)}_{\text{no. of times i'th layer is expanded}} \times \underbrace{b^i}_{\text{no. of nodes in i'th layer}}$



Iterative Deepening vs. BFS - example

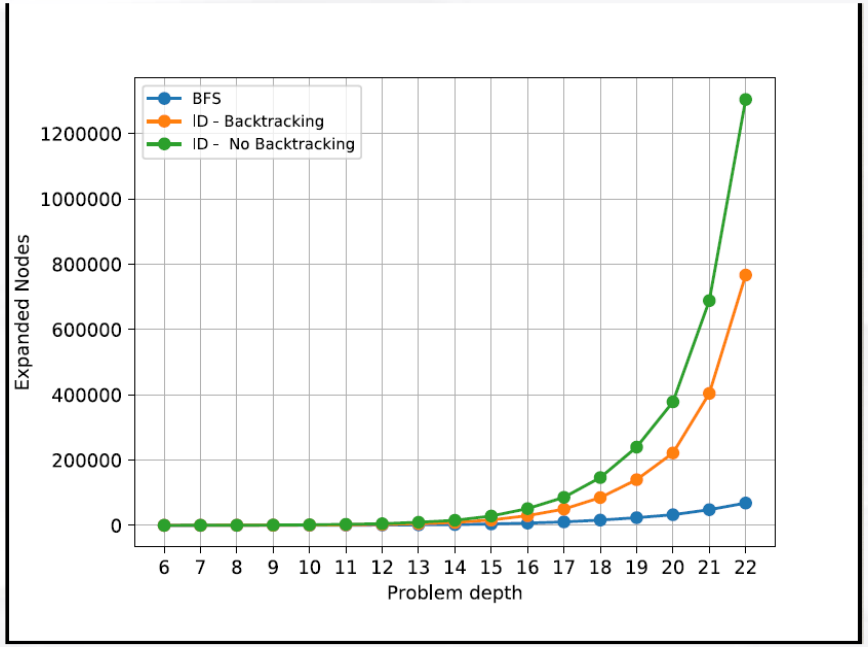
- Assume that **b=4** and **L=20**
- **Time (BFS):** $\sum_{i=0}^{20} 4^i = \frac{4^{21}-1}{4-1} = 1,466,015,503,701$
- **Time (ID):** $\sum_{i=0}^{20} (20+1-i) \cdot 4^i = 1,954,687,338,261$
- **Memory (BFS):** equals to Time (BFS) i.e., $1,466,015,503,701$
- **Memory(ID):** $4 \cdot 20 = 80$



- **Ratio –**
  - $\frac{\text{Time(ID)}}{\text{Time(BFS)}} = 1.333$ $\quad$ $\frac{\text{Memory(BFS)}}{\text{Memory(ID)}} = 1.8 \cdot 10^{10}$
  - => reduction by a factor of $1.8 \cdot 10^{10}$ in memory for **33%** more time

Empirical demonstration

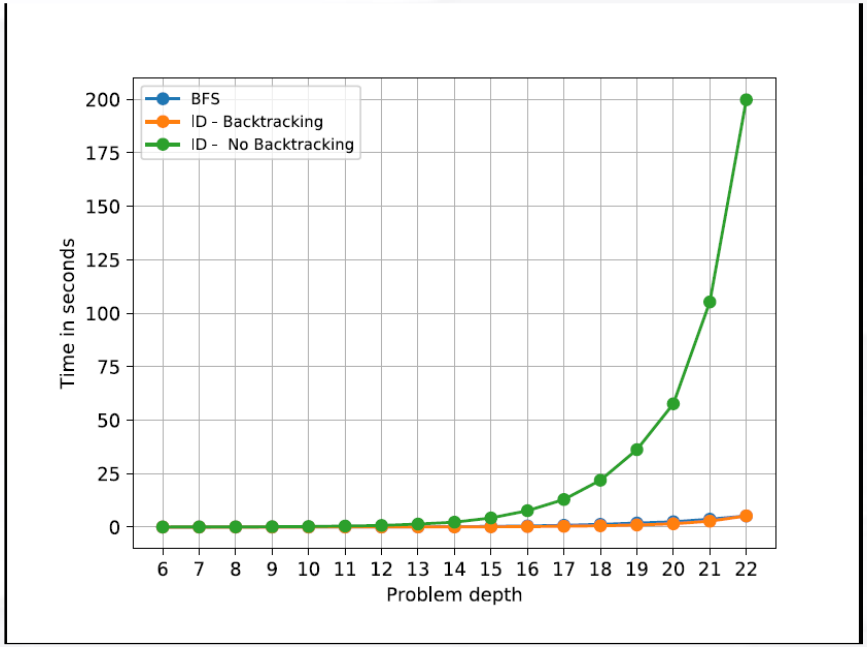
- **Expanded nodes** as a function of **problem depth**



- `ID` expands more nodes because
  - It repeats expanding nodes in different depth levels
  - It repeats expanding nodes because it does not perform a graph search (i.e., eliminating expanding duplicate nodes)

Empirical demonstration

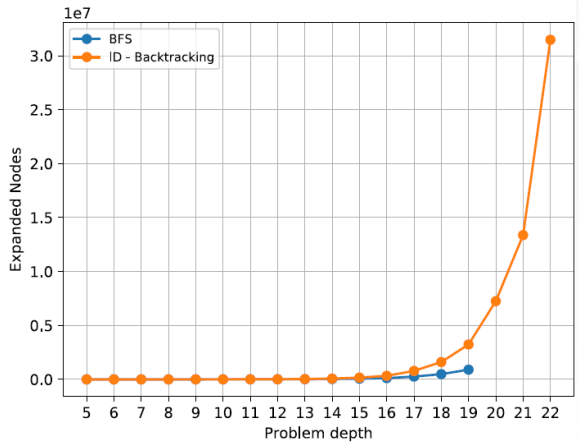
- **Time** as a function of **problem depth**



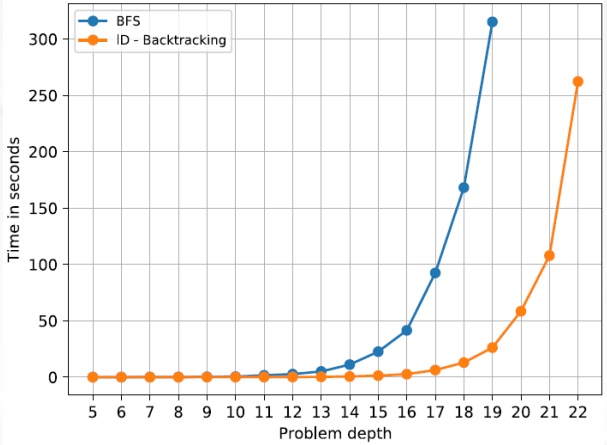
- `ID` requires much more time due to the additional number of nodes expanded
- `ID-BT` is much more efficient and has running time comparable to `BFS`

Empirical demonstration – 4X4 puzzle, BFS vs ID-BT

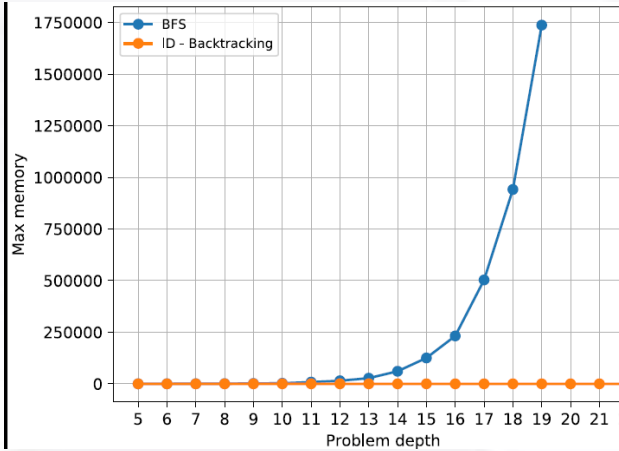
Expanded nodes



Time



Memory

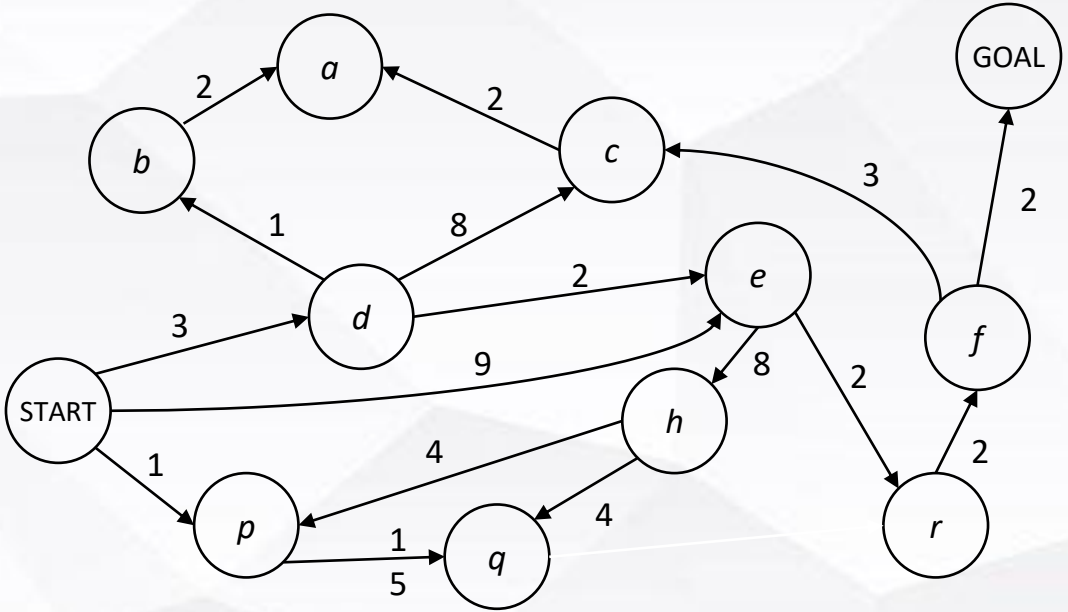


Uniform Cost Search

TECHNION
Israel Institute of Technology

The Henry and Marilyn Taub
Faculty of Computer Science

Cost-Sensitive Search

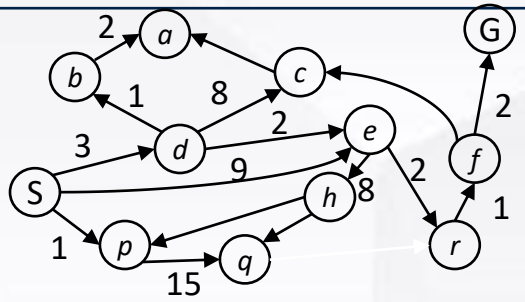


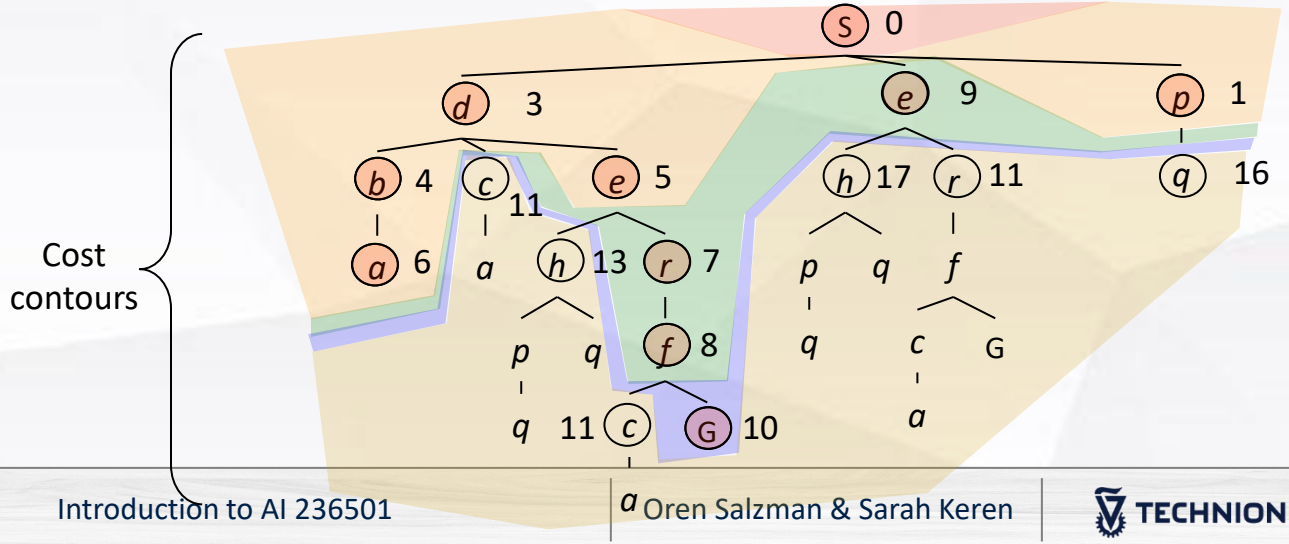
BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path.
We will now cover a similar algorithm which does find the least-cost path.

Uniform Cost Search

Strategy: expand a cheapest node first:

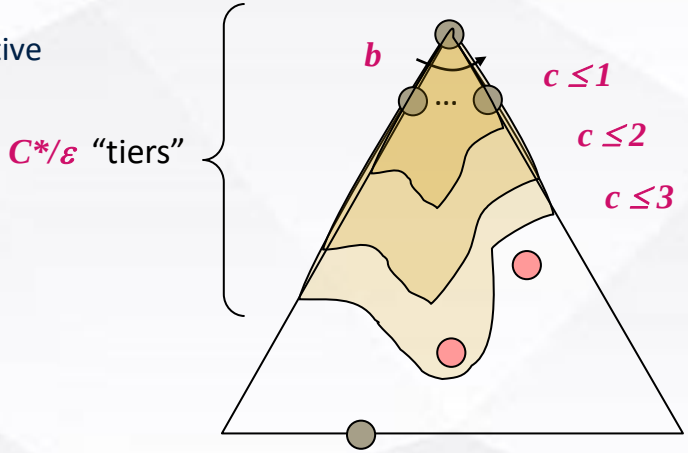
Fringe is a priority queue (priority: cumulative cost)





Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



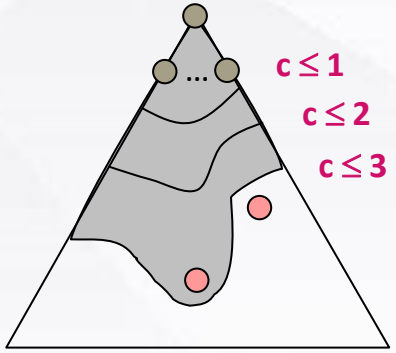
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!

- Is it optimal?
  - Yes! (Proof next lecture via `A*`)

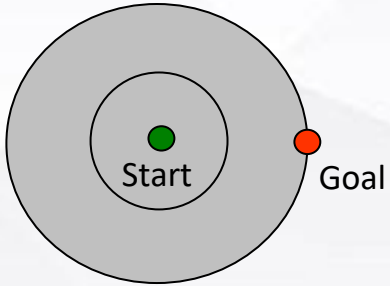
Uniform Cost Issues

- Remember: UCS explores increasing cost contours



- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location



- We’ll fix that soon!

The One Queue

- `BFS` – FIFO Queue
- `DFS` – LIFO Queue (Stack)
- `UCTS` – Priority Queue => **log(n)** overhead

- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are **priority queues** (i.e. collections of nodes with attached priorities)
  - Practically, for `DFS` and `BFS`, you can avoid the **log(n)** overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object

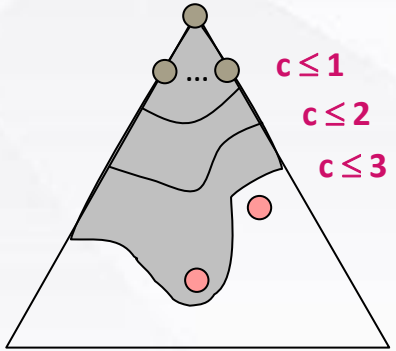
General Tree Search (recap)

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

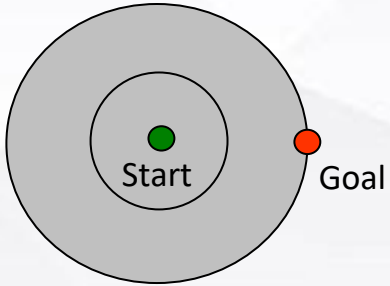
Uniform Cost Issues

- Remember: UCS explores increasing cost contours



- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location



- We’ll fix that soon!

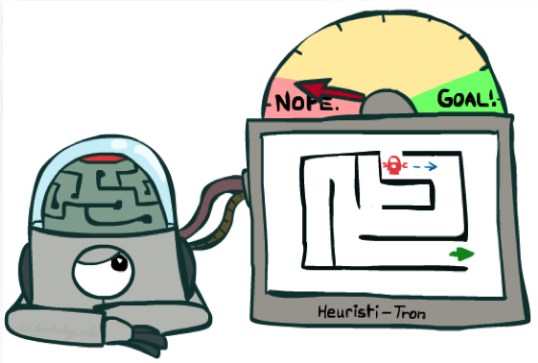
Informed Search

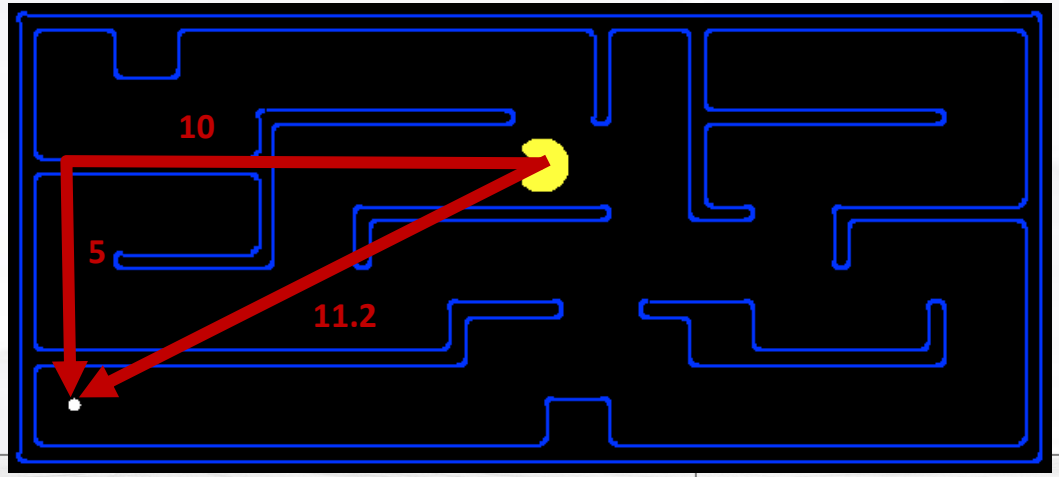
TECHNION
Israel Institute of Technology

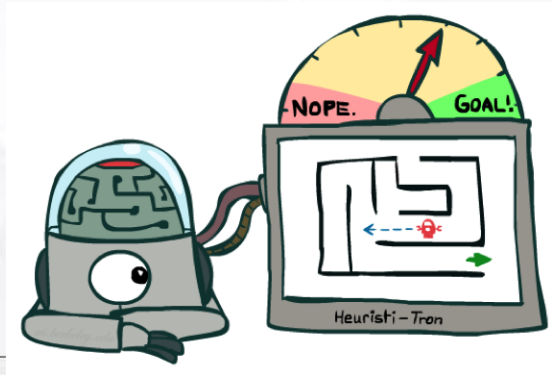
The Henry and Marilyn Taub
Faculty of Computer Science

Search Heuristics

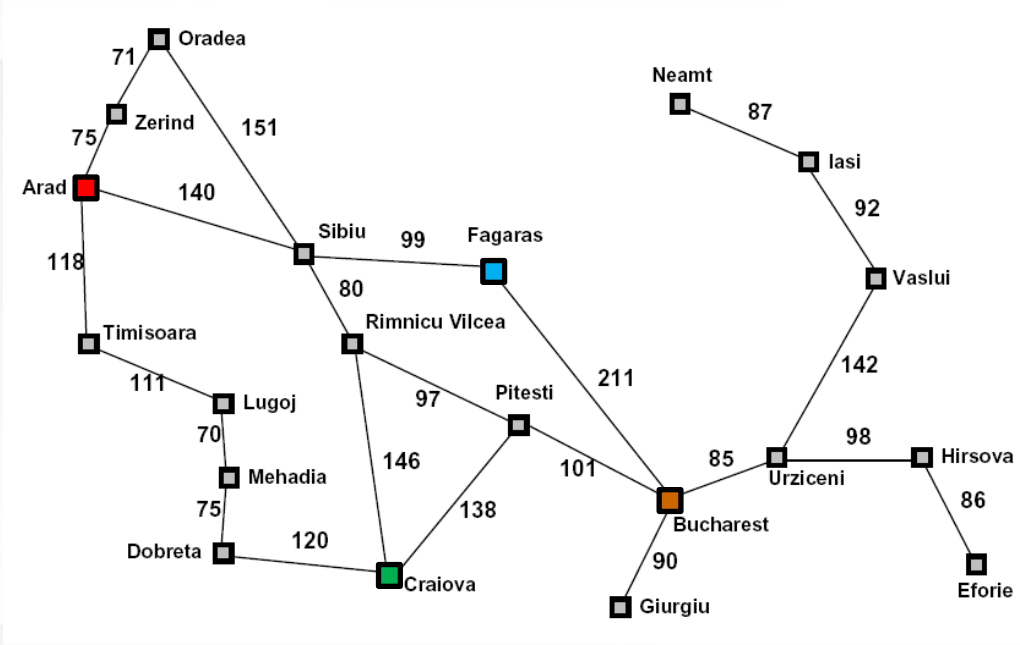
- A heuristic is:
  - A function that **estimates** how close a state is to a goal
  - Designed for a particular search problem
  - Examples (path finding): Manhattan distance, Euclidean distance







Example: Heuristic Function



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

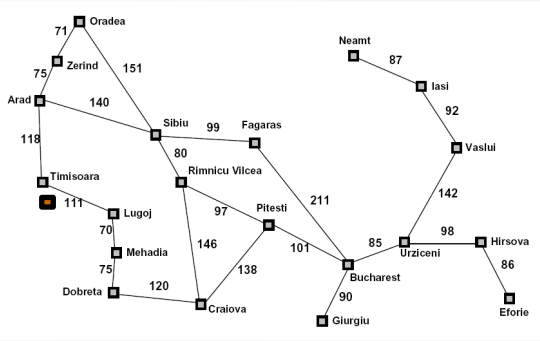
Greedy best-first Search

TECHNION
Israel Institute of Technology

The Henry and Marilyn Taub
Faculty of Computer Science

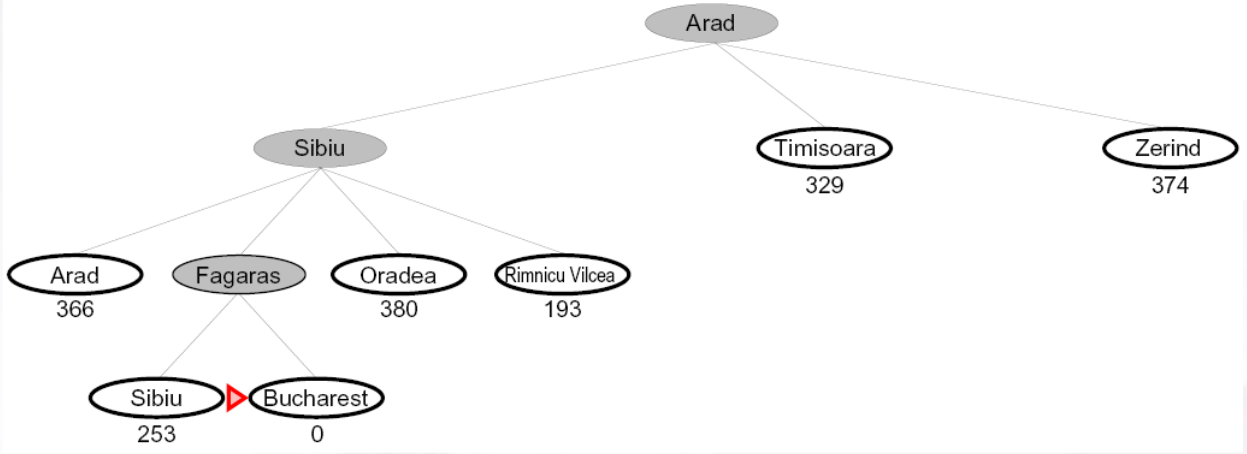
Greedy Search

- Expand the node that seems closest…



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)



- Is it optimal?
  - No. Resulting path to Bucharest is not the shortest!

Greedy Search – pseudo code

`Greedy best first search` (search problem - **P**)

- OPEN <- make_node(**P.start, NIL, h**(**P.start**)) //order according to **h**-value
- CLOSE <- {}
- **While** OPEN $\neq \emptyset$
  - **n** <- OPEN.pop_min()
  - CLOSE <- CLOSE $\cup$ {**n**}
  - **If P.goal_test**(**n**)
    - **Return** path(**n**)
  - For **s** in **P.SUCC** (**n**)
    - **If s** $\notin$ OPEN $\cup$ CLOSED
      - **n'** <- make_node(**s**, **n**, **h**(**s**))
      - OPEN.insert(**n'**)
- **Return failure**

A node consists of:

- State
- Pointer to parent
- h-value

Greedy best-first search - properties

- In finite spaces the algorithm is complete
- Not complete in infinite state spaces
  - Can be **arbitrarily bad**
- Possibly non-optimal solutions
- Time and space complexity depends on heuristic
  - Proportional to the number of nodes in OPEN and CLOSED
  - Could be much larger or smaller than **`BFS`**

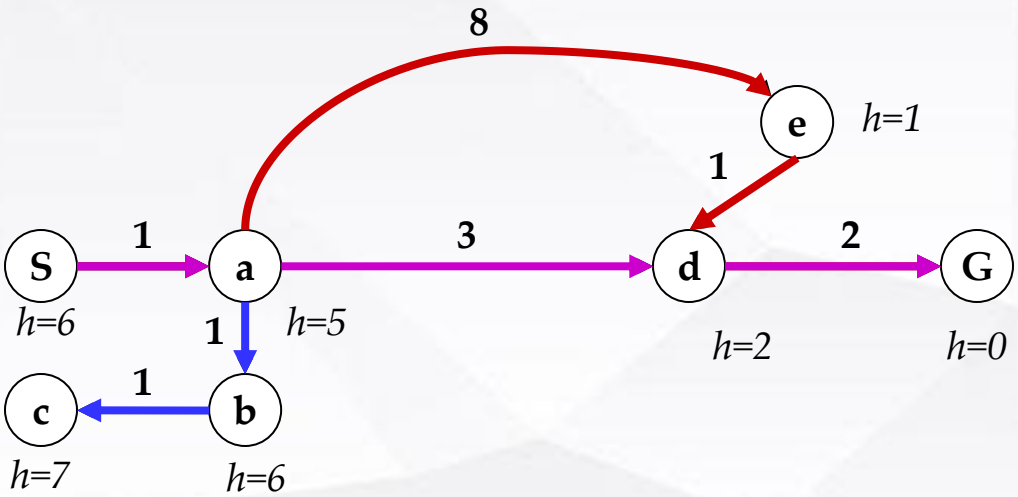
A*

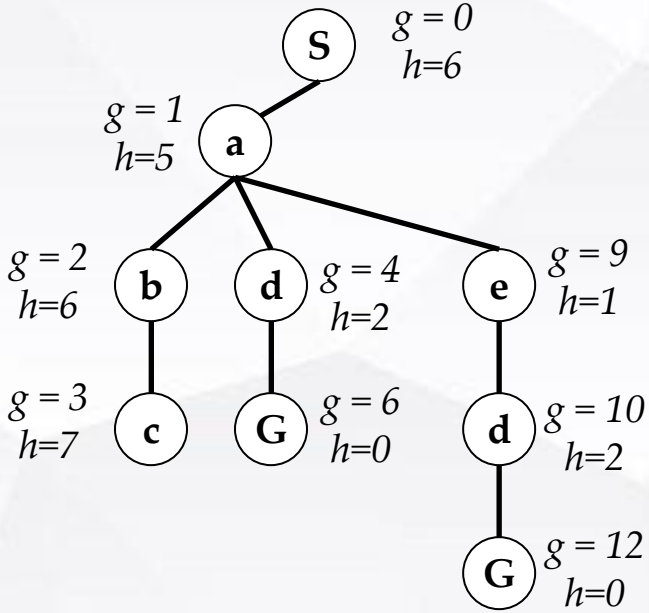
TECHNION
Israel Institute of Tech

The Henry and Marilyn Taub
Faculty of Computer Science

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* **g(n)** (aka cost-to-come)
- Greedy orders by goal proximity, or *forward cost* **h(n)** (aka cost-to-go)





- A* Search orders by the sum: **f(n) = g(n) + h(n)**

A* Search – pseudo code (1/2)

`A* search` (search problem - **P**)

- OPEN <- make_node(**P.start, NIL, 0, h**(**P.start**)) //order according to **f**-value
- CLOSE <- {}
- **While** OPEN $\neq \emptyset$
  - **n** <- OPEN.pop_min()
  - CLOSE <- CLOSE $\cup$ {**n**}
  - **If P.goal_test**(**n**)
    - **Return** path(**n**)
  - For **s** in **P.SUCC** (**n**)
    - **new_g <- g**(n) + **P.COST**(**n.state**,**s**) //newly-computed cost to reach **s**
    - **If s** $\notin$ OPEN $\cup$ CLOSED
      - **n'** <- make_node(**s**, **n**, **new_g, h(s)**)
      - OPEN.insert(**n'**)

A node consists of:

- State
- Pointer to parent
- **g**-value
- **f**-value

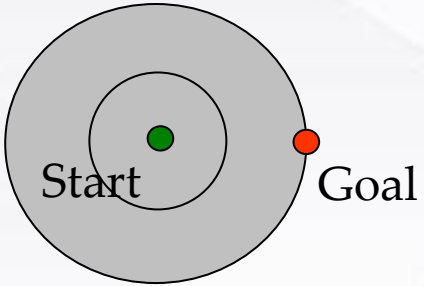
A* Search – pseudo code (2/2)

- **Else if s** ∈ OPEN
  - **n_curr** <- node in OPEN with state **s**
  - **If new_g < g(n_curr)** //found better path to **s**
    - **n_curr** <- update_node(**s**, **n**, **new_g** , **h**(**s**))
    - OPEN.update_key(**n_curr**) //don't forget to update place in OPEN…

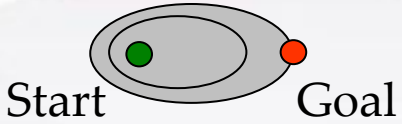
      //else do nothing – existing path is better
- **Else** // **s** CLOSED
  - **n_curr** <- node in CLOSED with state **s**
  - **If new_g < g(n_curr)** //found better path to **s**
    - **n_curr** <- update_node(**s**, **n**, **new_g** , **h**(**s**))
    - OPEN.insert(**n_curr**)
    - CLOSED.remove(**n_curr**)
- **Return** failure

UCS vs A* Contours

- Uniform-cost expands equally in all "directions"

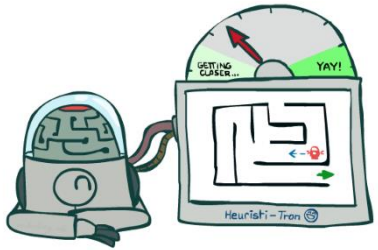


- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



A* - optimality (admissible heuristics)

TECHNION
Israel Institute of Technology



The Henry and Marilyn Taub
Faculty of Computer Science

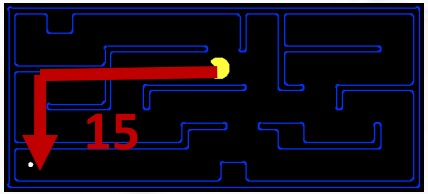
Admissible Heuristics

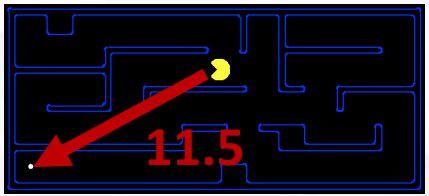
- A heuristic ***h*** is ***admissible*** (optimistic) if:

$$\forall x \; 0 \leq h(x) \leq h^*(x)$$

where $h^*(x)$ is the true cost to a nearest goal

- Examples:





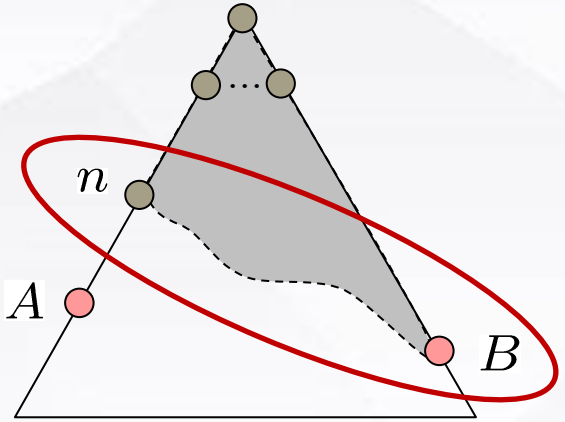
0.0

- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A* Tree Search: Blocking

Proof:

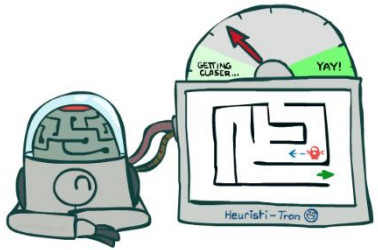
- Imagine **B** is on the fringe
- Some ancestor **n** of **A** is on the fringe, too (maybe **A**!)
- **Claim**: **n** will be expanded before **B**
  1. **f(n)** is less or equal to **f(A)**
  2. **f(A)** is less than **f(B)**
  3. **n** expands before **B**
- All ancestors of **A** expand before **B**
- **A** expands before **B**
- A* search is optimal (with an admissible heuristic)



$$f(n) \leq f(A) < f(B)$$

Efficiency of A* (heuristics quality)

TECHNION
Israel Institute of Technology



The Henry and Marilyn Taub
Faculty of Computer Science

Efficiency of A*

- The efficiency of A* depends on the heuristic quality
- What happens when $h \equiv 0$?
- What happens when $h \equiv h^*$ (optimal heuristic)?

Efficiency of A*

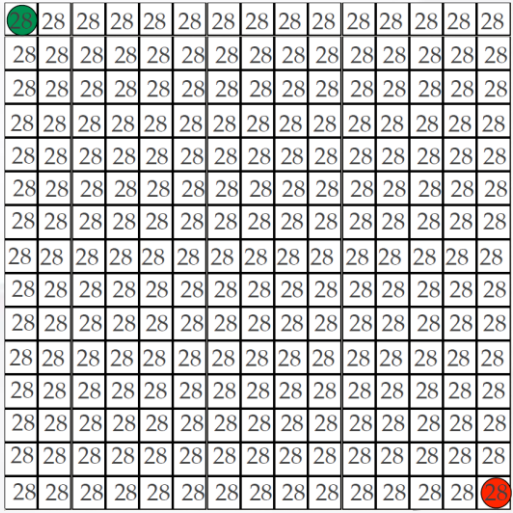
- The efficiency of A* depends on the heuristic quality
- What happens when $h \equiv 0$?
  - Uses only **g**-values => identical to Uniform-cost search
- What happens when $h \equiv h^*$ (optimal heuristic)?

Efficiency of A*

- The efficiency of `A*` depends on the heuristic quality
- What happens when $h \equiv 0$?
  - Uses only **g**-values => identical to `Uniform-cost search`
- What happens when $h \equiv h^*$ (optimal heuristic)?
  - Will only expand nodes along an optimal path (prove at home)
  - What is the difference between this setting and `greedy-best first` with a perfect heuristic?
  - What if there is more than one optimal path?

Efficiency of A*

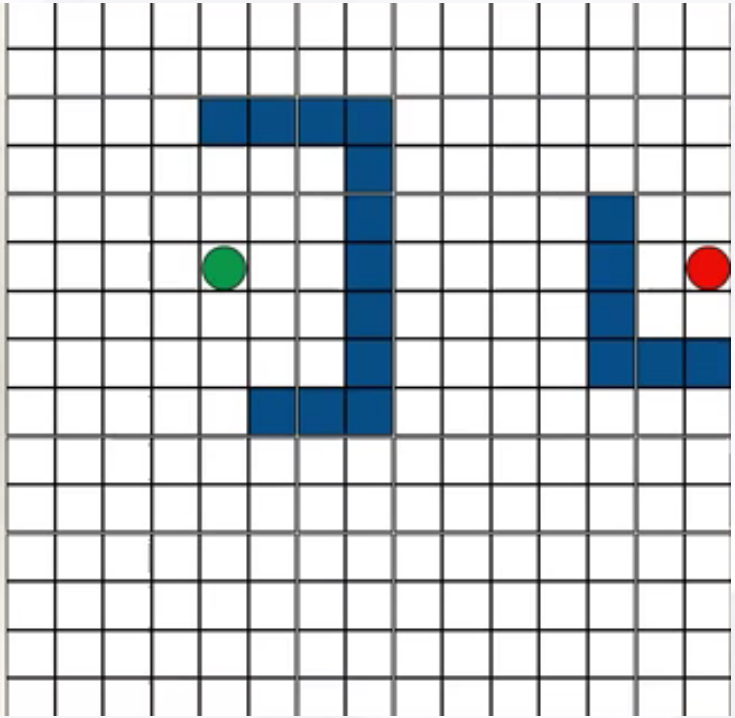
- The efficiency of A* depends on the heuristic quality
- What happens when $h \equiv 0$?
  - Uses only **g**-values => identical to Uniform-cost search
- What happens when $h \equiv h^*$ (optimal heuristic)?
  - Will only expand nodes along an optimal path (prove at home)
  - What is the difference between this setting and greedy-best first with a perfect heuristic?
  - What if there is more than one optimal path?



Efficiency of A*

- What happens in the general case?
  - An informative heuristic allows ignoring paths that cannot be part of an optimal path
  - Typically, the more informative the heuristic is the more focused the search can be
  - Sometimes, computing a highly-informative heuristics takes more time than using a simple (but effective) one

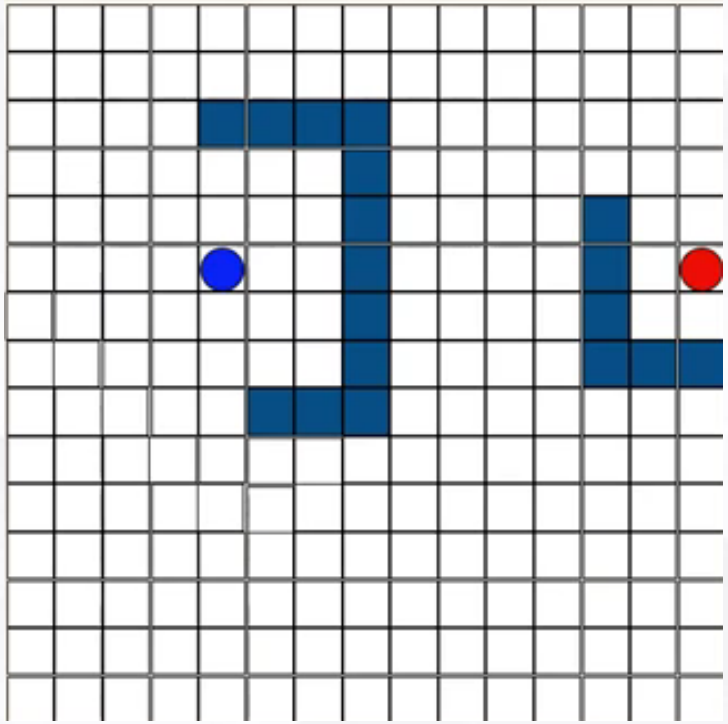
Greedy Best-first Search Demo



Visualization by Shaul Markovitz, video available at https://youtu.be/n1Cm1XiGd48

Greedy best-first search demo

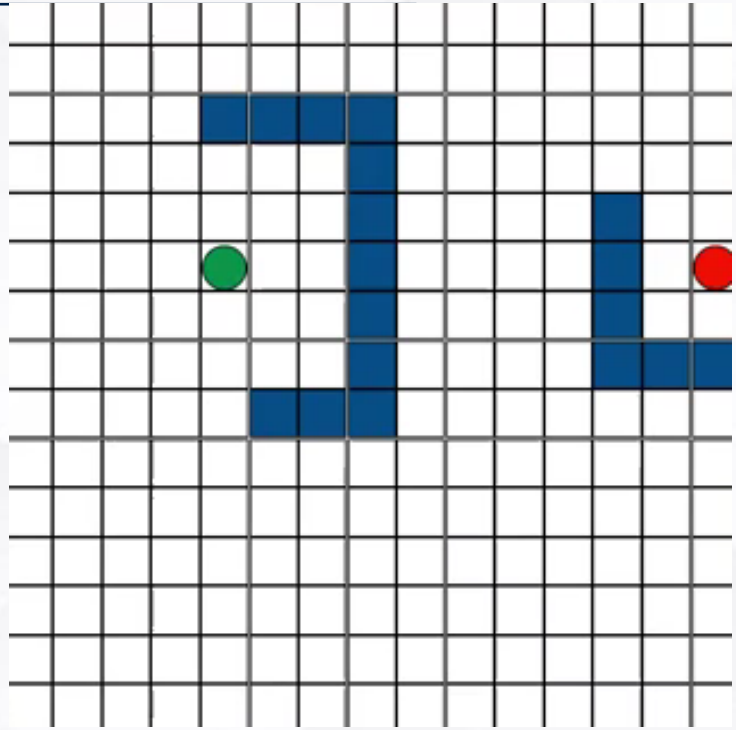
- Heuristic: Manhattan distance



Visualization by Shaul Markovitz, video available at https://youtu.be/TdHbO3w68fY

A* demo

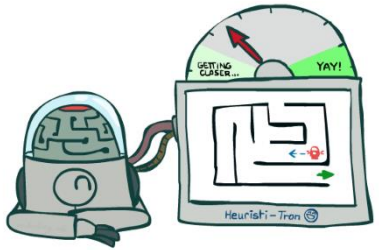
- Heuristic: Manhattan distance



Visualization by Shaul Markovitz, video available at https://youtu.be/F6HQ_KzWuYQ

A*
(consistent heuristics)

TECHNION
Israel Institute of Technology



The Henry and Marilyn Taub
Faculty of Computer Science

Consistency of Heuristics

- A heuristic function is called **consistent** (also called **monotonic**) if

$$\forall s \in S,\ \forall s' \in \mathrm{SUCC}(s)\ h(s) - h(s') \leq \mathrm{COST}(s, s')$$

- Roughly speaking a consistent heuristic is not only **globally optimistic** (like admissible heuristics) but also **locally optimistic**
  - Let's re-write the definition: $h(s') + \mathrm{COST}(s, s') \geq h(s)$

Consistency of Heuristics

- A heuristic function is called **consistent** (also called **monotonic**) if

$$\forall s \in S,\ \forall s' \in \mathrm{SUCC}(s)\ h(s) - h(s') \leq \mathrm{COST}(s, s')$$

- Roughly speaking a consistent heuristic is not only **globally optimistic** (like admissible heuristics) but also **locally optimistic**
  - Let's re-write the definition: $h(s') + \mathrm{COST}(s, s') \geq h(s)$
  - Now, let's show that the f-values monotonically increase during the search

$$f(s') = g(s') + h(s')$$

Consistency of Heuristics

- A heuristic function is called **consistent** (also called **monotonic**) if

$$\forall s \in S,\ \forall s' \in \mathrm{SUCC}(s)\ h(s) - h(s') \leq \mathrm{COST}(s, s')$$

- Roughly speaking a consistent heuristic is not only **globally optimistic** (like admissible heuristics) but also **locally optimistic**
  - Let's re-write the definition: $h(s') + \mathrm{COST}(s, s') \geq h(s)$
  - Now, let's show that the f-values monotonically increase during the search

$$f(s') = g(s') + h(s') = g(s) + COST(s, s') + h(s')$$

Consistency of Heuristics

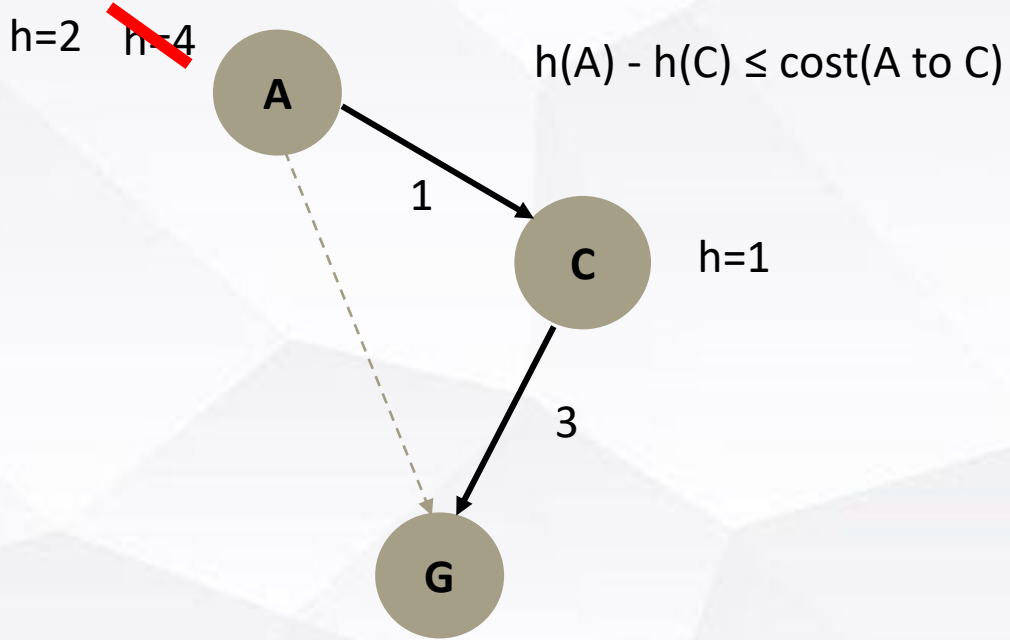
- A heuristic function is called **consistent** (also called **monotonic**) if

$$\forall s \in S,\ \forall s' \in \mathrm{SUCC}(s)\ h(s) - h(s') \leq \mathrm{COST}(s, s')$$

- Roughly speaking a consistent heuristic is not only **globally optimistic** (like admissible heuristics) but also **locally optimistic**
  - Let's re-write the definition: $h(s') + \mathrm{COST}(s, s') \geq h(s)$
  - Now, let's show that the f-values monotonically increase during the search

$$f(s') = g(s') + h(s') = g(s) + COST(s, s') + h(s') \geq g(s) + h(s) = f(s)$$

Consistency of Heuristics



h(A) - h(C) ≤ cost(A to C)

Is h consistent ?

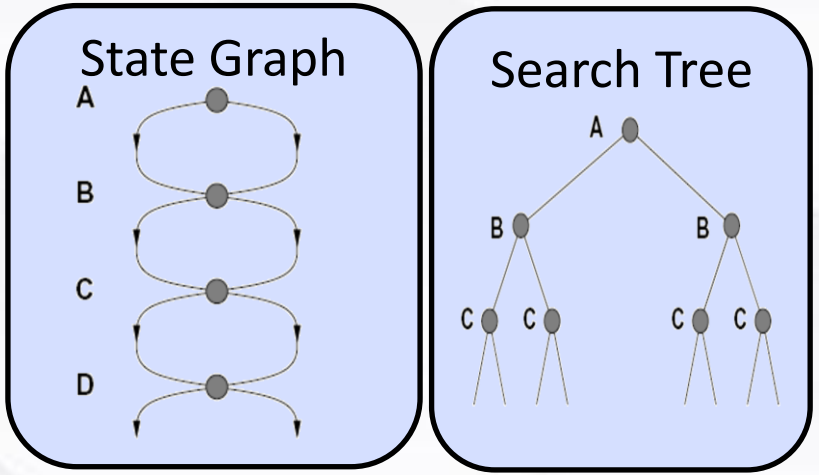
Consistency of Heuristics

- When A* uses a consistent heuristic the path to every node that was expanded is optimal (proof in Pearl's book)
- Namely, $\forall n \in \text{CLOSED}\ g(n) = g^*(n)$

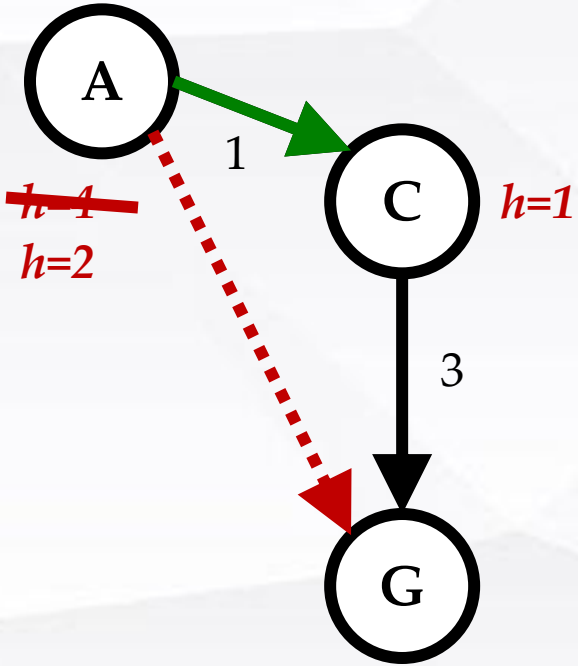
  $\forall s \in CLOSED\colon\ g(s) = g^*(s)$
- => When using a consistent heuristic, no need to move nodes back from CLOSED to OPEN

Closed list (actually not a list but a set…)

- Used to avoid expanding a state twice
  - Failure to detect repeated states can cause exponentially more work
- But why do we need to move nodes from the CLOSED list to the OPEN list?
  - Because the heuristic mislead us…



Admissibility and 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 admissibility :
  - A* search is optimal
- Consequences of consistency:
  - The **f**-value along a path never decreases

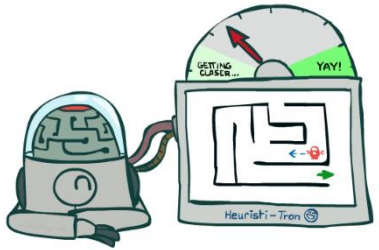
    h(A) ≤ cost(A to C) + h(C)

Optimality of A*'s efficiency

- When A* uses a consistent heuristic it can be shown that any other (optimal) search algorithm that uses the same set heuristic expands at least the same number of nodes
- Is that what we always care about?

Computing heuristics

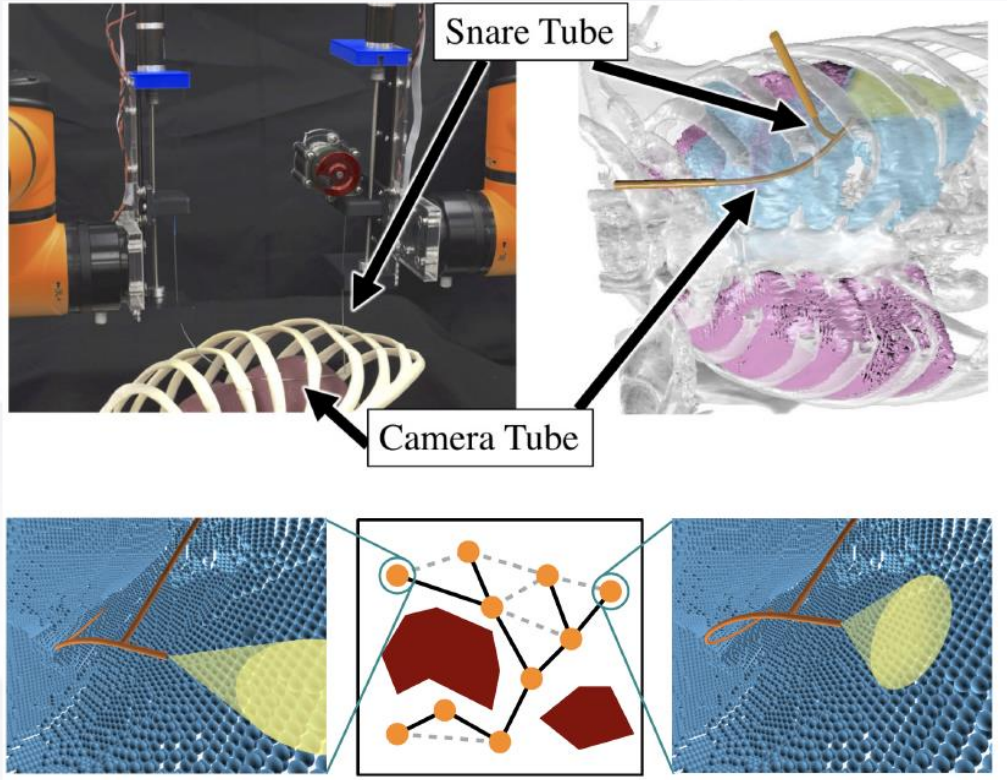
TECHNION
Israel Institute of Technology



The Henry and Marilyn Taub
Faculty of Computer Science

A* Applications

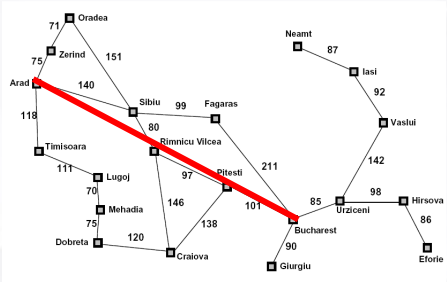
- Video games
- Path finding / routing problems
- Resource-planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

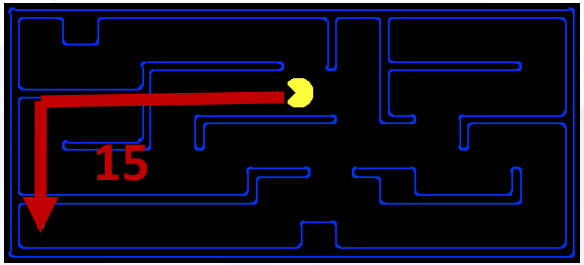


Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
- Often, admissible heuristics are solutions to *relaxed problems*, where new actions are available

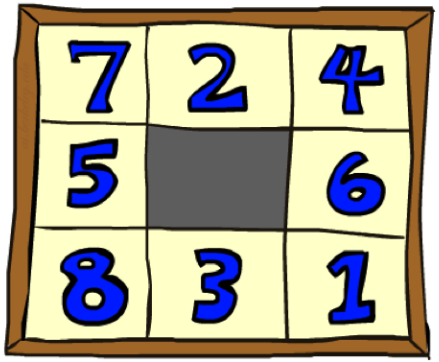
366



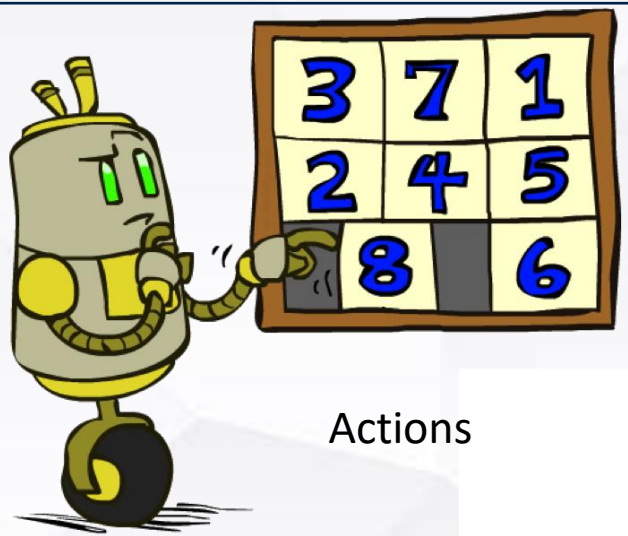


- Inadmissible heuristics are often useful too

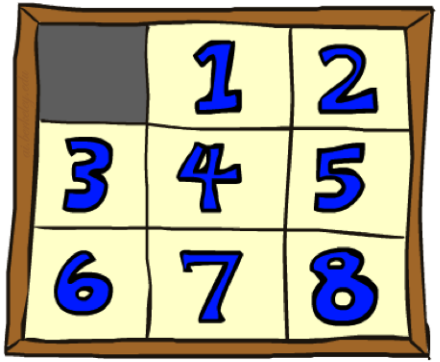
Example: 8 Puzzle



Start State



Actions



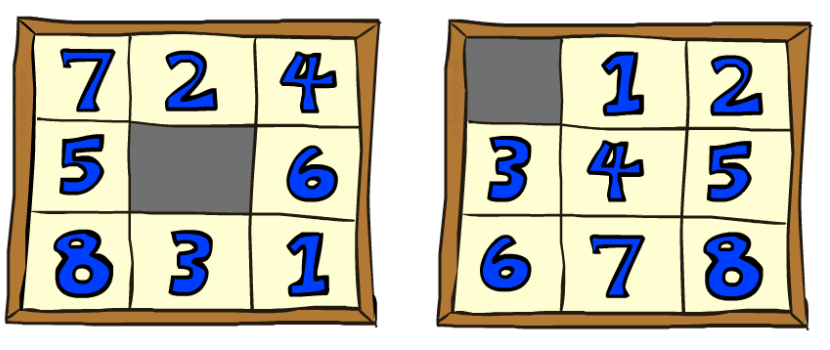
Goal State

- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

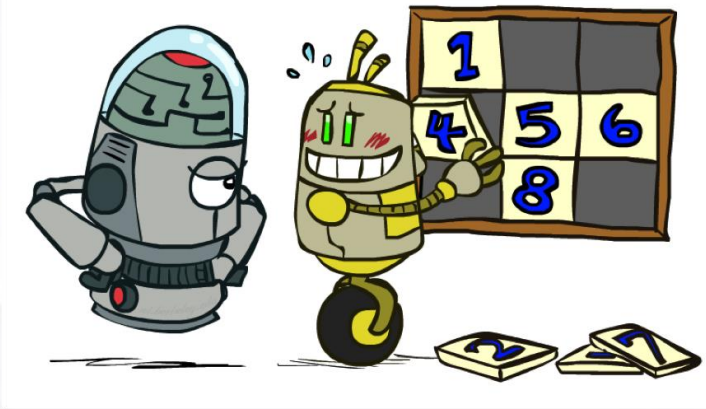
Admissible heuristics?

8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = **8**
- This is a *relaxed-problem* heuristic



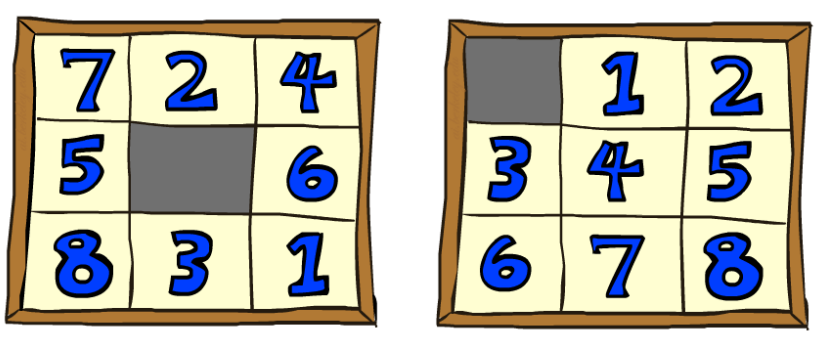
Start State Goal State



	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
A* + TILES	13	39	227

8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhattan* distance
- Why is it admissible?
- h(start) = 3 + 1 + 2 + … = 18

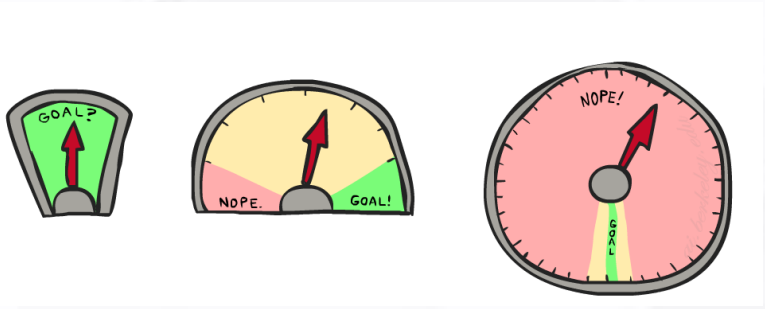


Start State Goal State

	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
A* + TILES	13	39	227
A* + MANHATTAN	12	25	73

8 Puzzle III

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?



- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

Bounded suboptimal search
Trading quality for efficiency

TECHNION
Israel Institute of Technology

The Henry and Marilyn Taub
Faculty of Computer Science

Trading quality for efficiency in a structured manner

- The A* algorithm computes an optimal solution
- Sometimes we are willing to compromise slightly on the quality of the solution if we can get it faster
- An algorithm is said to be **bounded suboptimal** if the quality of the solution is bounded by a multiplicative factor when compared to the optimal solution

Weighted A*

- Exactly like A* but given a user-provided $w \in [0,1]$, orders nodes in OPEN according to

$$f(n) = (1-w) \cdot g(n) + w \cdot h(n)$$

- What happens when
  - **w=0**?
  - **w=1**?
  - **w=0.5**?
  - **w<0.5**?

Weighted A*

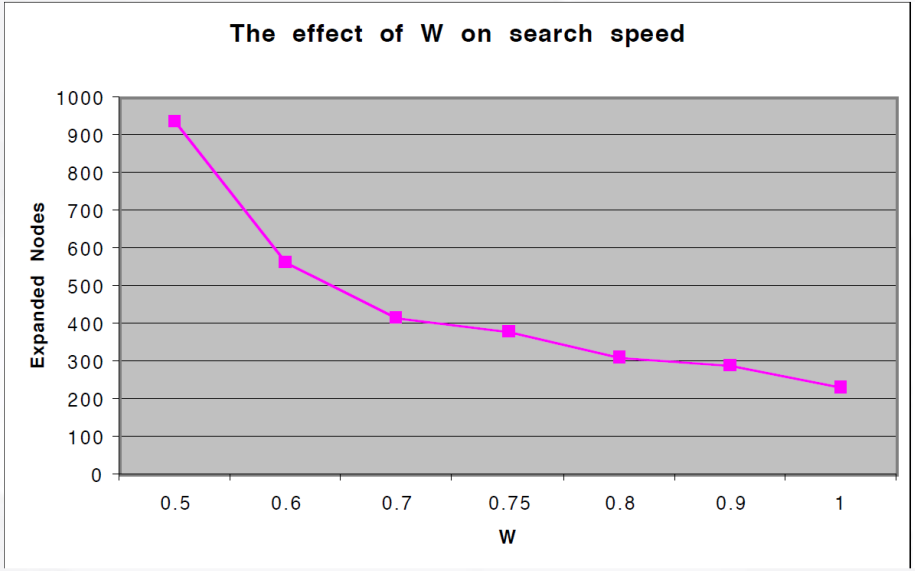
- Exactly like `A*` but given a user-provided $w \in [0,1]$, orders nodes in OPEN according to

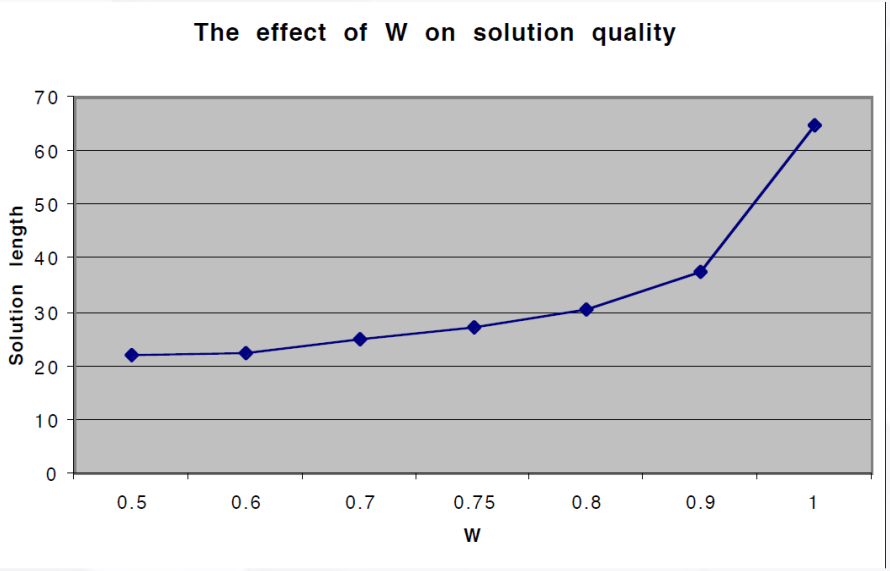
$$f(n) = (1-w) \cdot g(n) + w \cdot h(n)$$

- What happens when
  - **w=0**? `UCS`
  - **w=1**? `Greedy Best First Search`
  - **w=0.5**? `A*`
  - **w<0.5**?

Empirical demonstration of wA*

- Every point is the average result of 100 random instances of the alg. On a 3X3 puzzle using the Manhattan heuristic





Weighted A* (alternative formulation)

- Alternative formulation: given a user-provided $w \geq 1$, orders nodes in OPEN according to $f(n) = g(n) + w \cdot h(n)$
- **Thm**: wA* (under this formulation) returns a solution whose cost is at most $w \cdot C^*$ where, C^* is the cost of the optimal solution

A*-epsilon

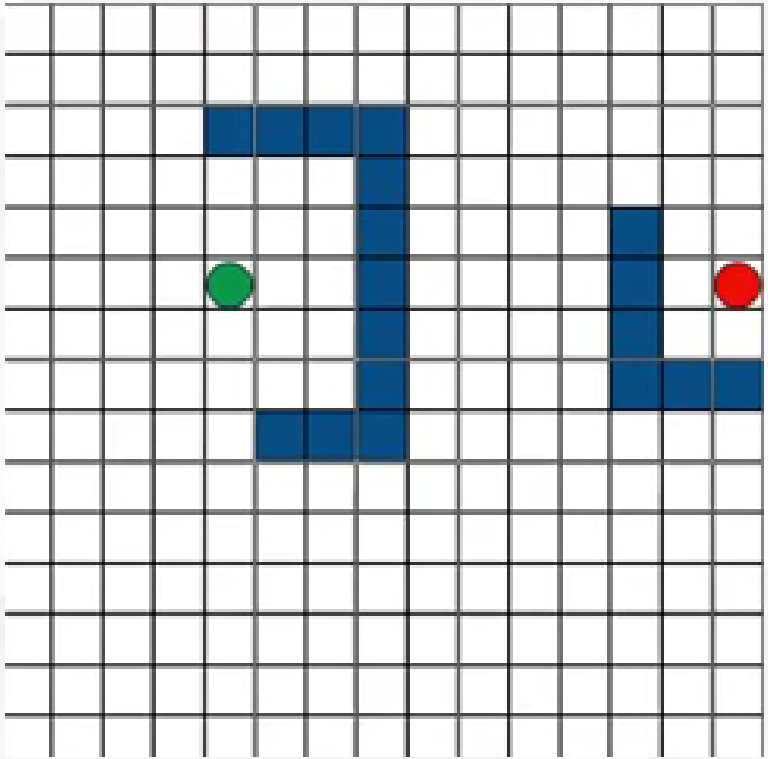
- Similar to A* in addition to a heuristic function h we are also given
  - a user-provided approximation factor $\varepsilon \geq 0$
  - a (possibly-inadmissible) heuristic function h_{focal}
- In A* we expand a node with the minimal **f**-value
- Here, we
  - Compute the set of "almost best nodes" according to h

$$\text{FOCAL} = \{n \in \text{OPEN} \mid f(n) \leq (1+\varepsilon) \cdot \min_{n' \in \text{OPEN}} f(n')\}$$

  - Choose a node according to h_{focal} with $\min_{n \in \text{FOCAL}} h_{\text{FOCAL}}(n)$

A*-epsilon

- Using $\varepsilon = 0.1$



A*-epsilon – bounded suboptimality

- **Thm**: A*-epsilon returns a solution whose cost is at most $(1+\varepsilon)\cdot C^*$ where, C^* is the cost of the optimal solution under the condition that h is admissible.
- **Proof**: Let
  - n_0 be the node in OPEN with the minimal f-value (has to be in FOCAL)
  - t be the node returned by A*-epsilon when terminated
  - n_ℓ be a node in OPEN that belongs to the optimal path with all ancestors in CLOSED
  - The cost of the solution returned is $C(t) = f(t) = g(t)$

A*-epsilon – bounded suboptimality (cont.)

- Note that
  - $f(n_\ell) \leq C^*$ since the heuristic is admissible
  - $f(n_0) \leq f(n_\ell)$ by definition of n_0
  - $f(t) \leq (1+\varepsilon) \cdot f(n_0)$ since t was chosen from FOCAL

- Thus,

$$C(t) = f(t) \leq (1+\varepsilon) \cdot f(n_0) \leq (1+\varepsilon) \cdot f(n_\ell) \leq (1+\varepsilon) \cdot C^*$$

IDA*

TECHNION
Israel Institute of Technology

The Henry and Marilyn Taub
Faculty of Computer Science

IDA*

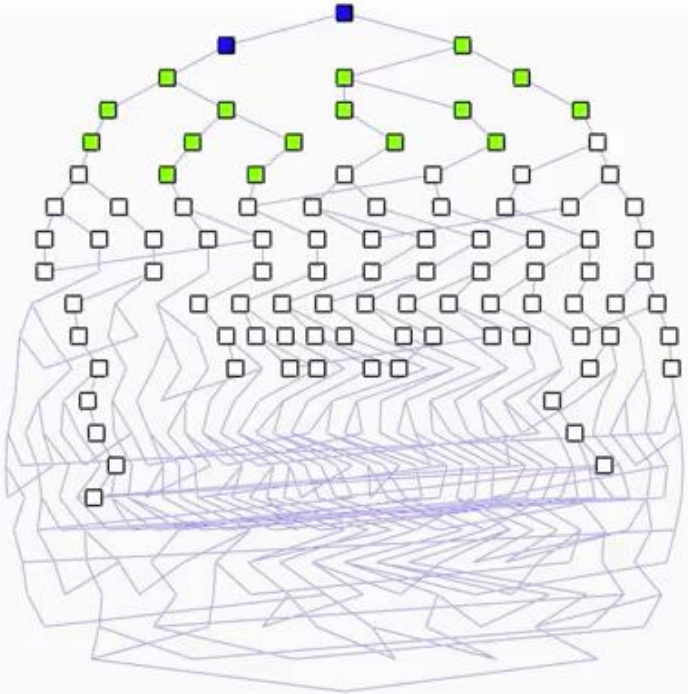
- One of the main drawbacks of best-first search algorithms (like `A*`) is there large memory footprint (due to the OPEN list)
- Iterative deepening `A*` (`IDA*`) uses **f**-values to **bound the depth** of the search it performs
- It starts with a small bound on the maximal **f**-values and increases it between iterations
  - Start with a bound of $f \leq h(s_{\text{start}})$
  - Bound in the next iteration is the minimal **f**-value found in the current iteration (that is larger than the existing bound)

IDA* Search – pseudo code

```
function Iterative-Deepening-A* (problem):
        new_limit ← h(problem.init_state)
        While Not Interrupted:
                f_limit ← new_limit
                new_limit ← ∞ /* Global variable */
                result ← DFS-f (problem.init_state, 0, null, f_limit, problem)
                if result ≠ failure then return result
        return failure
```

```
function DFS-f (state, g, path, f_limit, problem):
        new_f ← g + h(state)
        if new_f > f_limit:
                new_limit ← min(new_limit, new_f)
                return failure
        if problem.goal(state) then return path
        For s in expand(state):
                result ← DFS-f (s, g + cost(state, s), path || s, f_limit, problem)
                if result ≠ failure then return result
        return failure
```

Visualization (2X3 sliding puzzle)



```
Current Limit: 0
Next Limit: 11

Current f-cost: 11 (g: 0 h: 11)
```

Video adapted from
https://movingai.com/

Summary

Algorithm		Complete	Optimal	Time	Space
Blind	Breadth First Search	Y	Y*	O(b^m)	O(b^m)
	Depth First Search	N**	N	O(b^m)	O(mb)
	Iterative Deepening	Y	Y	O(b^m)	O(mb)
	Uniform Cost Search	Y***	Y***	O(b^m)	O(b^m)
Informed	Greedy Best First Search	N**	N	O(b^m)	O(b^m)
	A*	Y***+^	Y***+^	O(b^m)	O(b^m)
	IDA*	Y***+^	Y***+^	O(b^m)	O(mb)

* When edge costs are one

** Y when finite graph with no cycles

*** When costs are positive and lower bounded

^ When j is admissible