AlphaGO

- Tic-Tac-Toe — PhD project @ Cambridge 1952
- Chess — Deep Blue 1997
- Jeopardy — Watson 2011
- Atari Games from raw pixels (Google) — 2014
- AlphaGo — Deepmind — 2016
  - GO game search space?
    - more than a GooGol larger than Chess!!
      - greater than #atoms in universe
Search Space

- 1,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000 possible positions
AlphaGo

- Monte Carlo Tree Search
  - **Policy Network** — suggest moves
  - **Value Network** — estimates eventual winner

- Deep Neural Networks
  - 12 layers (supervised and reinforcement learning)
  - millions of connections
  - trained with 30 million of moves (human experts)
  - predicted 57%
  - networks by itself defeat state-of-the-art programs (based on trees)
“On our chip we bring the data as close as possible to the processing units, and move the data as little as possible,”

“Whereas many of the cores in a GPU share a single, large memory bank, each of the Eyeriss cores has its own memory.”

Eyeriss has 168 processing elements (PE), each with its own memory.
Today

- Informed Search
  - A* Search (Analysis)
- Graph Search
A* Search
Combining UCS and Greedy

Example: Teg Grenager
Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* $g(n)$

Example: Teg Grenager
Combining UCS and Greedy

- Uniform-cost orders by path cost, or \textit{backward cost} $g(n)$

Example: Teg Grenager
Combining UCS and Greedy

- **Uniform-cost** orders by path cost, or *backward cost* \( g(n) \)

---

Example: Teg Grenager
Combining UCS and Greedy

- **Uniform-cost** orders by path cost, or *backward cost* $g(n)$
- **Greedy** orders by goal proximity, or *forward cost* $h(n)$

Example: Teg Grenager
Combining UCS and Greedy

- **Uniform-cost** orders by path cost, or *backward cost* $g(n)$
- **Greedy** orders by goal proximity, or *forward cost* $h(n)$

Example: Teg Grenager
Combining UCS and Greedy

- **Uniform-cost** orders by path cost, or *backward cost* $g(n)$
- **Greedy** orders by goal proximity, or *forward cost* $h(n)$

Example: Teg Grenager
Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Greedy orders by goal proximity, or *forward cost* $h(n)$

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

---

Example: Teg Grenager
Combining UCS and Greedy

- **Uniform-cost** orders by path cost, or *backward cost* $g(n)$
- **Greedy** orders by goal proximity, or *forward cost* $h(n)$

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

Example: Teg Grenager
Is A* Optimal?

What went wrong?

- [Diagram of a graph with nodes S, A, and G, and edges with weights and heuristic values.]
  - S to A: 1, h = 7
  - A to G: 5, h = 6
  - S to G: 3, h = 0
Is A* Optimal?

- What went wrong?
- Actual bad goal cost < estimated good goal cost
Is A* Optimal?

- What went wrong?
  - Actual bad goal cost < estimated good goal cost
  - We need estimates to be less than actual costs!
A heuristic $h$ is **admissible** (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal
Admissible Heuristics

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

$$0 \leq h(n) \leq h^*(n)$$

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

- Example:
A heuristic $h$ is \textit{admissible} (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

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

Example:
A heuristic $h$ is **admissible** (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

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

Example:

Coming up with admissible heuristics is most of what’s involved in using $A^*$ in practice.
Optimality of A* Tree Search
Optimality of A* Tree Search

Assume:
- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:
- A will exit the fringe before B
Optimality of A* Tree Search: Blocking

Proof:
Optimality of A* Tree Search: Blocking

Proof:
- Imagine B is on the fringe
Proof:
- Imagine B is on the fringe
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!)
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!)
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
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
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 \( f(A) \)
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 $f(A)$
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 $f(A)$

$$f(n) = g(n) + h(n)$$  
Definition of f-cost
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 $f(A)$

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

Definition of f-cost

$$f(n) \leq g(A)$$

Admissibility of h
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 \( f(A) \)

\[
f(n) = g(n) + h(n) \\
f(n) \leq g(A) \\
g(A) = f(A)
\]

Definition of f-cost
Admissibility of h
\( h = 0 \) at a goal
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 $f(B)$
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 $f(B)$
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 $f(B)$
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 $f(B)$

\[ g(A) < g(B) \quad \text{B is suboptimal} \]
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 $f(B)$
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
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
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

$f(n) \leq f(A) < f(B)$
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
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
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
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
Semi-Lattice of Heuristics
Trivial Heuristics, Dominance

- Dominance: $h_a \geq h_c$ if
  \[ \forall n : h_a(n) \geq h_c(n) \]
- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible
    \[ h(n) = \max(h_a(n), h_b(n)) \]
- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic
Graph Search
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.
- In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)
Graph Search

- In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)
Graph Search

- Idea: never expand a state twice
Graph Search

- Idea: never expand a state twice

- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
Graph Search

- Idea: never expand a state twice

- How to implement:
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

- Important: store the closed set as a set, not a list
Graph Search

- **Idea:** never expand a state twice

- **How to implement:**
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

- **Important:** store the closed set as a set, not a list

- Can graph search wreck completeness? Why/why not?
Graph Search

- **Idea:** never expand a state twice

- **How to implement:**
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

- **Important:** store the closed set as a set, not a list

- Can graph search wreck completeness? Why/why not?

- How about optimality?
Graph Search

- Idea: never expand a state twice

- How to implement:
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set

- Important: store the closed set as a set, not a list

- Can graph search wreck completeness? Why/why not?

- How about optimality?
A* Graph Search Gone Wrong?

State space graph
A* Graph Search Gone Wrong?

State space graph

![State space graph](image-url)
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)
A* Graph Search Gone Wrong?

State space graph

Search tree

- **S (0+2)**
- **A (1+4)**
- **B (1+1)**
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)
A (1+4)  B (1+1)
  ↓      ↓
  C (3+1)
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)

A (1+4)

B (1+1)

C (3+1)

G (6+0)
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)

A (1+4)  B (1+1)

C (2+1)  C (3+1)

G (6+0)

S

B

C

G
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)

A (1+4) B (1+1)

C (2+1) C (3+1)

G (6+0)
A* Graph Search Gone Wrong?

State space graph

Search tree

- S (0+2)
  - A (1+4)
    - C (2+1)
      - G (5+0)
  - B (1+1)
    - C (3+1)
      - G (6+0)
A* Graph Search Gone Wrong?

State space graph

Search tree

S (0+2)

A (1+4)

B (1+1)

C (3+1)

G (6+0)
Consistency of Heuristics
Consistency of Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from } A \text{ to } G \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost}(A \text{ to } C) \]
Consistency of Heuristics

- **Main idea: estimated heuristic costs ≤ actual costs**
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from } A \text{ to } G \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost}(A \text{ to } C) \]

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
  - A* graph search is optimal
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - A* graph search is optimal
Consistency of Heuristics

- **Main idea**: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- **Consequences of consistency**:
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - A* graph search is optimal

\[ h=4 \]
Consistency of Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- Consequences of consistency:
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - A* graph search is optimal
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from } A \text{ to } G \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost}(A \text{ to } C) \]

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
  - A* graph search is optimal
## Consistency of Heuristics

- **Main idea:** estimated heuristic costs $\leq$ actual costs
  - **Admissibility:** heuristic cost $\leq$ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - **Consistency:** heuristic “arc” cost $\leq$ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost}(A \text{ to } C) \]

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
  - A* graph search is optimal
Consistency of Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- Consequences of consistency:
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - A* graph search is optimal
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - **Admissibility:** heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - **Consistency:** heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - \( A^* \) graph search is optimal
Consistency of Heuristics

- **Main idea:** estimated heuristic costs ≤ actual costs
  - **Admissibility:** heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - **Consistency:** heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- **Consequences of consistency:**
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - A* graph search is optimal
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
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
    end
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
end
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
  end
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
end
Consider what A* does:
- Expands nodes in increasing total f value (f-contours)
  Reminder: $f(n) = g(n) + h(n) = \text{cost to } n + \text{heuristic}$
- Proof idea: the optimal goal(s) have the lowest f value, so it must get expanded first
Consider what A* does:

- Expands nodes in increasing total f value (f-contours)
  Reminder: $f(n) = g(n) + h(n) = \text{cost to } n + \text{heuristic}$
- Proof idea: the optimal goal(s) have the lowest f value, so it must get expanded first

There's a problem with this argument. What are we assuming is true?
Optimality of A* Graph Search

Proof:

![Diagram of A* Graph Search with nodes G and G* connected by a path]

The diagram illustrates the optimality of A* Graph Search, where G* represents the optimal path and G represents the path found by A*. The shaded area indicates the explored part of the graph.
Optimality of A* Graph Search

Proof:

- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
Proof:

- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
Proof:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
Optimality of A* Graph Search

Proof:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
Optimality of A* Graph Search

Proof:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was popped
Optimality of A* Graph Search

Proof:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was popped
- $f(p) < f(n)$ because of consistency
Optimality of A* Graph Search

Proof:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because $n'$ is suboptimal
Proof:

- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n’$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n’$ was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n’)$ because $n’$ is suboptimal
- $p$ would have been expanded before $n’$
Optimality of A* Graph Search

Proof:
- New possible problem: some $n$ on path to $G^*$ isn’t in queue when we need it, because some worse $n'$ for the same state dequeued and expanded first (disaster!)
- Take the highest such $n$ in tree
- Let $p$ be the ancestor of $n$ that was on the queue when $n'$ was popped
- $f(p) < f(n)$ because of consistency
- $f(n) < f(n')$ because $n'$ is suboptimal
- $p$ would have been expanded before $n'$
- Contradiction!
A* Applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- ...