COL864: Special Topics in AI
Semester II, 2021-22

State Space Planning: A*

Rohan Paul

Outline

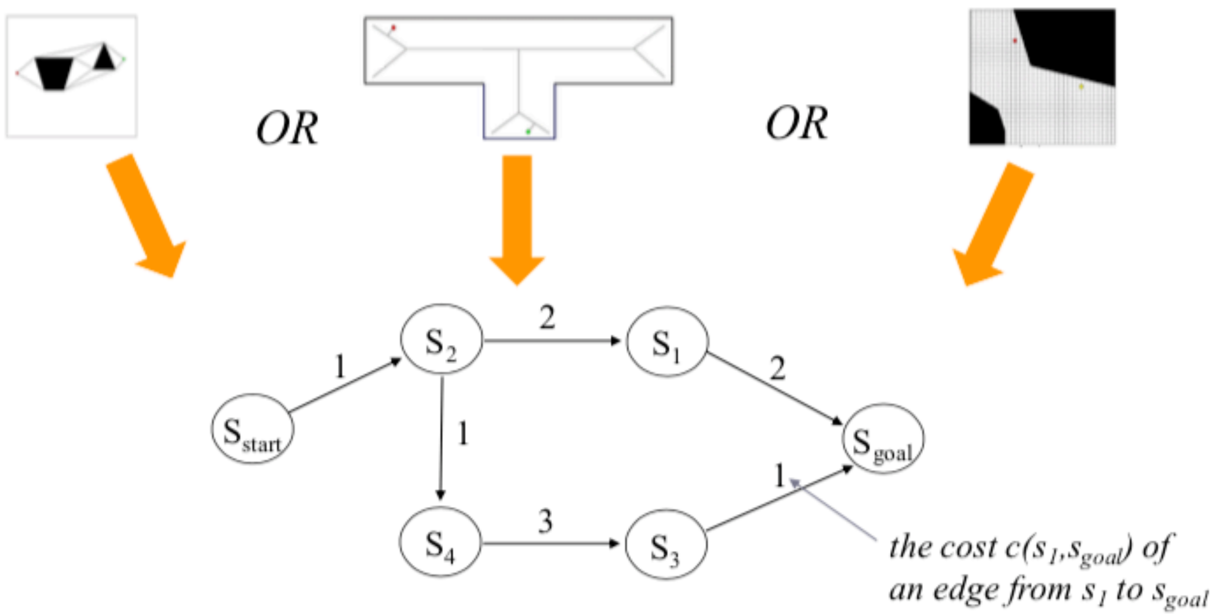
- Last Class
  - State Estimation
- This Class
  - Search Algorithms
    - Uninformed A*
    - Informed A* and extensions
- Reference Material
  - Primary reference are the lecture notes. For basic background refer to AIMA Ch. 3.

Acknowledgements

These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Nicholas Roy, Wolfram Burgard, Dieter Fox, Sebastian Thrun, Siddharth Srinivasa, Dan Klein, Pieter Abbeel, Max Likhachev and others.

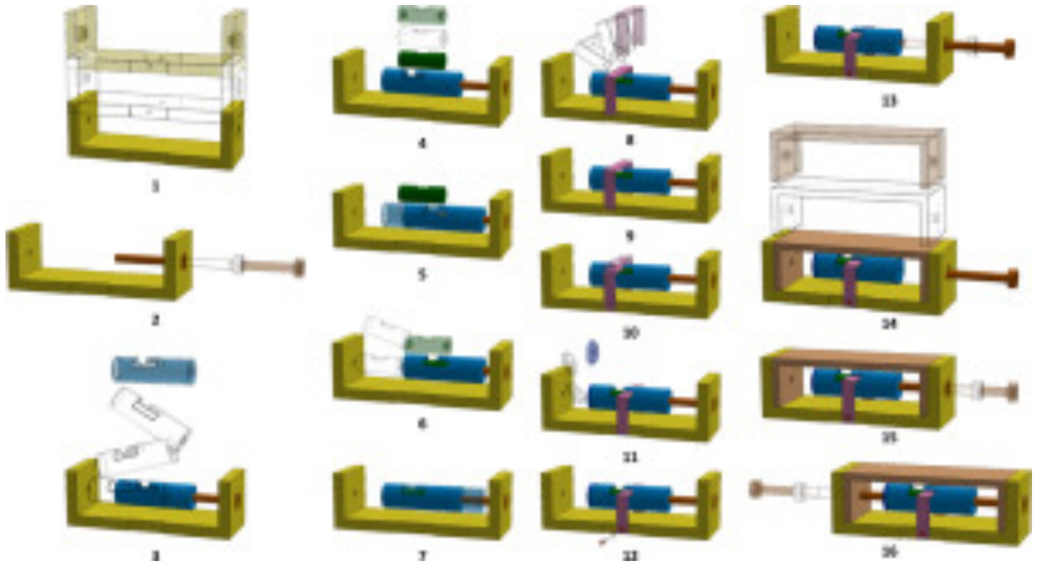
Planning with Graphs

- Planning graphs
  - Nodes: possible states (designated start and goal states)
  - Edges: connection between states if an action connects the two states.
  - Goal is to find the optimal path (sequences of actions.)
- Motion planning
  - A graph is constructed (from skeletonization or cell decomposition etc.)
  - Example: PRM or grids or some other decomposition of the space.
- Other planning problems
  - Task planning where pre-condition relationships exist between tasks.

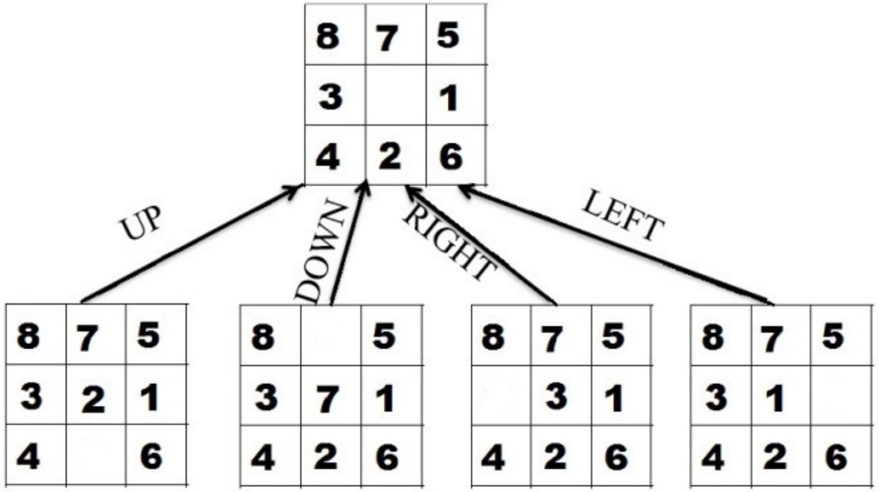


Applications

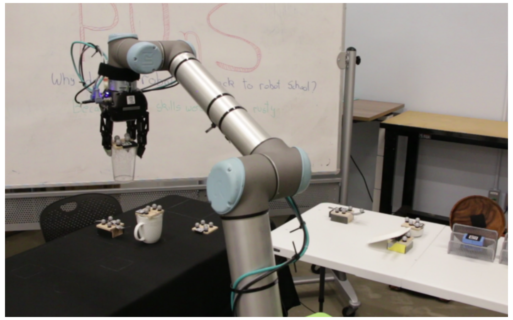
Assembly planning

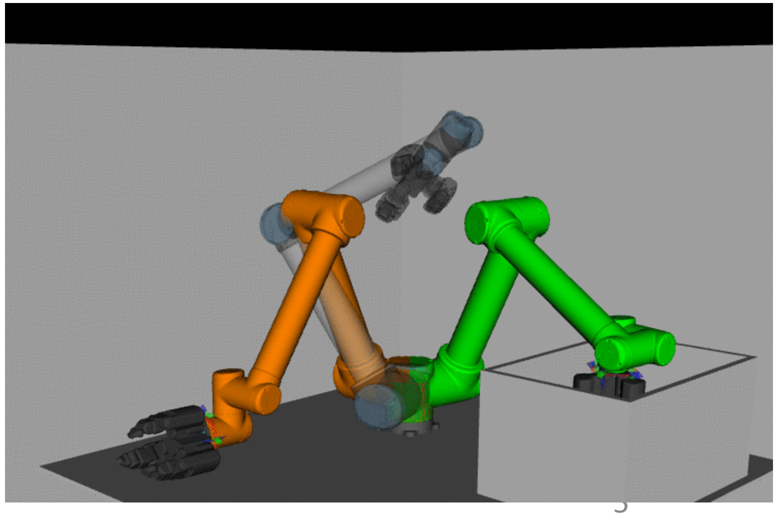


Tile puzzle



Complex motion planning

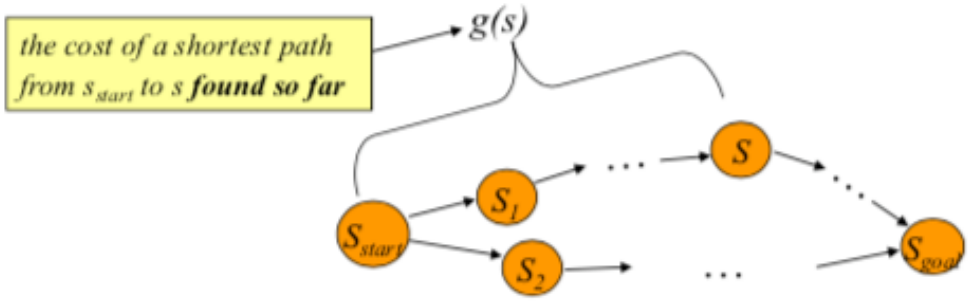


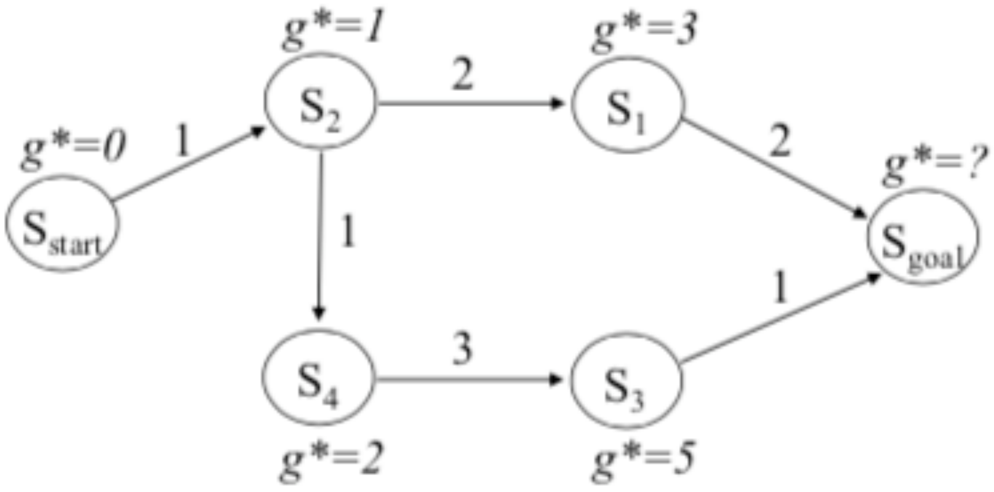


Searching Graphs for a Least-cost Path

- Important quantity
  - g*(s) – the cost of the least cost path from the start state to s.
  - Many search algorithms (including A*) work by computing g*(s) values for graph vertices (states).
  - The g*(s) values are the *"cost so far"* from the start state to the state s.
- Problem: how to determine $g^*(s_{goal})$?

– $g^*(s)$ – the cost of a least-cost path from s_{start} to s





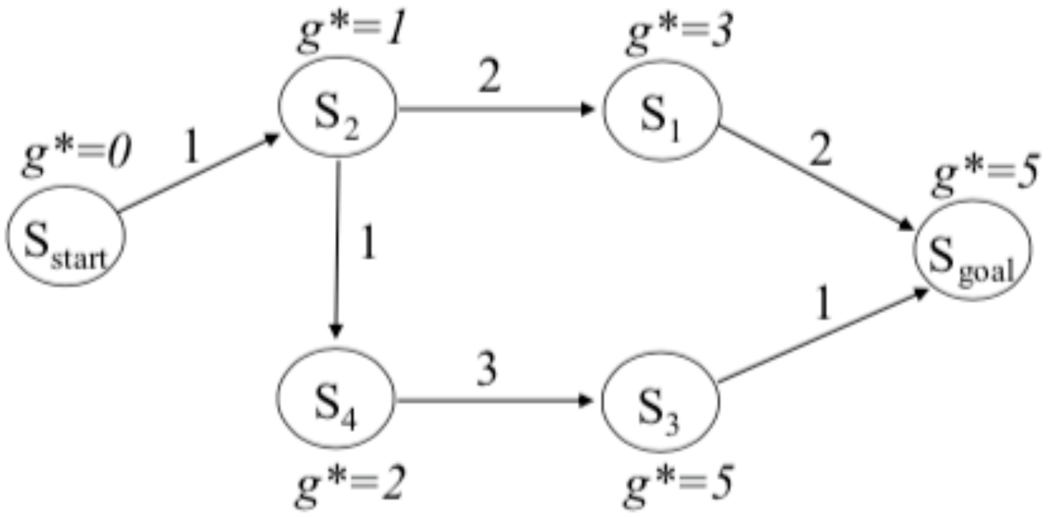
g*(s) values for nodes in a graph

Searching Graphs for a Least-cost Path

- The g*(s) values satisfy a recursive relationship.

$g^*(s)$ – the cost of a least-cost path from s_{start} to s

g* values satisfy: $g^*(s) = \min_{s'' \in pred(s)} g^*(s'') + c(s'',s)$

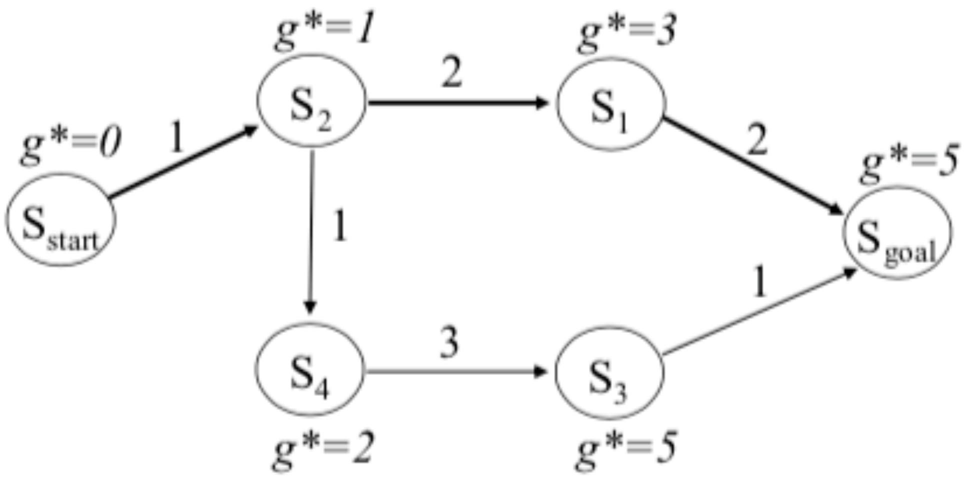


Searching Graphs for a Least-cost Path

- From g* values how to get the path?
  - First compute the g*-values are computed a least-cost path from s_{start} to s_{goal}
  - Then perform backtracking.

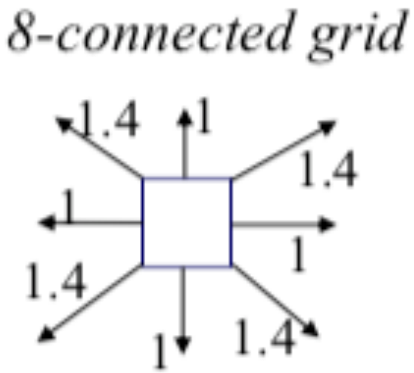
start with s_{goal} and from any state s backtrack to the predecessor state s' such that

$$s' = \arg\min_{s'' \in pred(s)} (g^*(s'') + c(s'', s))$$

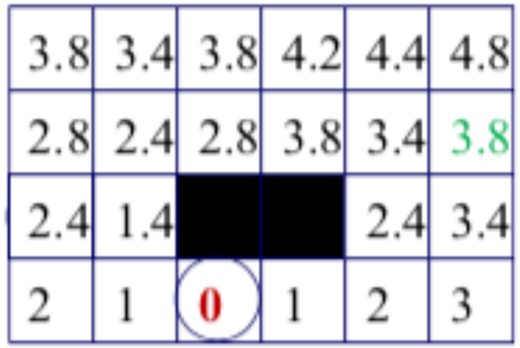


Searching Graphs for a Least-cost Path

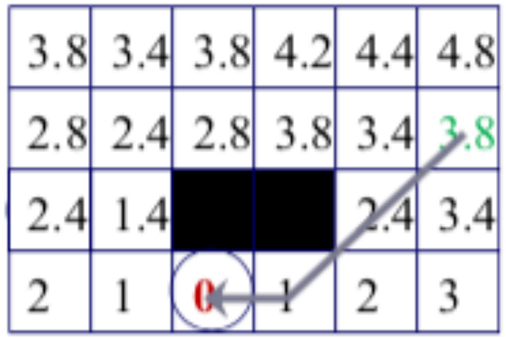
- Example: an agent in a grid-based graph
- Computing g*(s) values and then backtracking to get the path.



Actions and costs



g*(s) values for states in the grid



Path obtained via backtracking

Uninformed A* Search

Perform an operation on the graph to get the g*(s) values.

Main function

$g(s_{start}) = 0$; all other g-values are infinite; $OPEN = \{s_{start}\}$;

ComputePath();

publish solution; //compute least-cost path using g-values

set of candidates for expansion

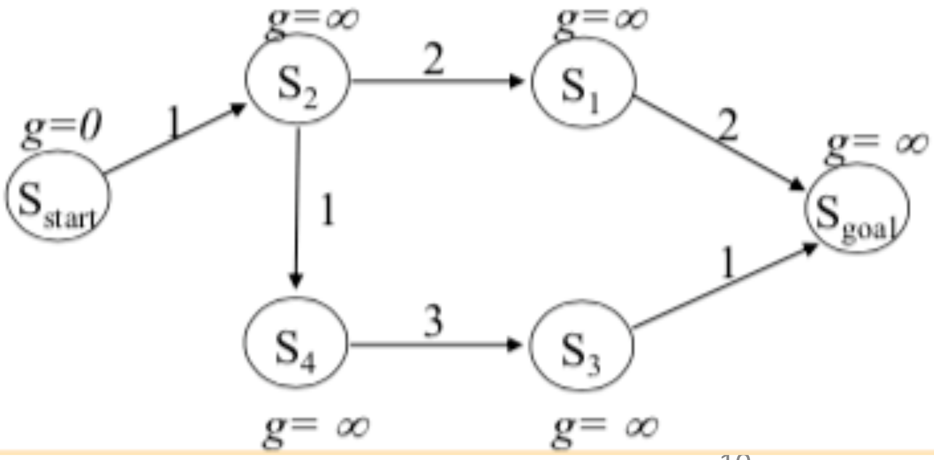
ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $g(s)$ from $OPEN$;

  expand s;

for every expanded state $g(s)$ is optimal ($g(s) = g^*(s)$)



Uninformed A* Search – cntd.

Main function

$g(s_{start}) = 0$; all other g-values are infinite; $OPEN = \{s_{start}\}$;

ComputePath();

publish solution; //compute least-cost path using g-values

set of candidates for expansion

ComputePath function

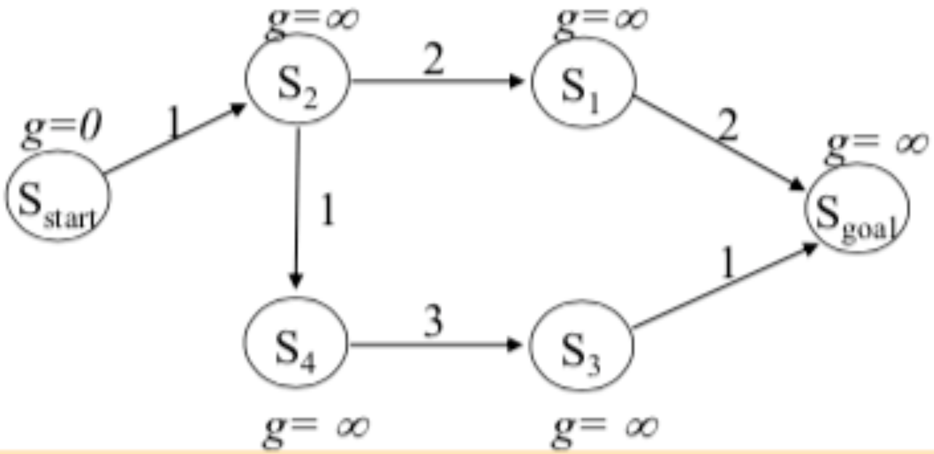
while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $g(s)$ from $OPEN$;

  expand s;

What is expansion?

for every expanded state $g(s)$ is optimal ($g(s) = g^*(s)$)



Uninformed A* Search – cntd.

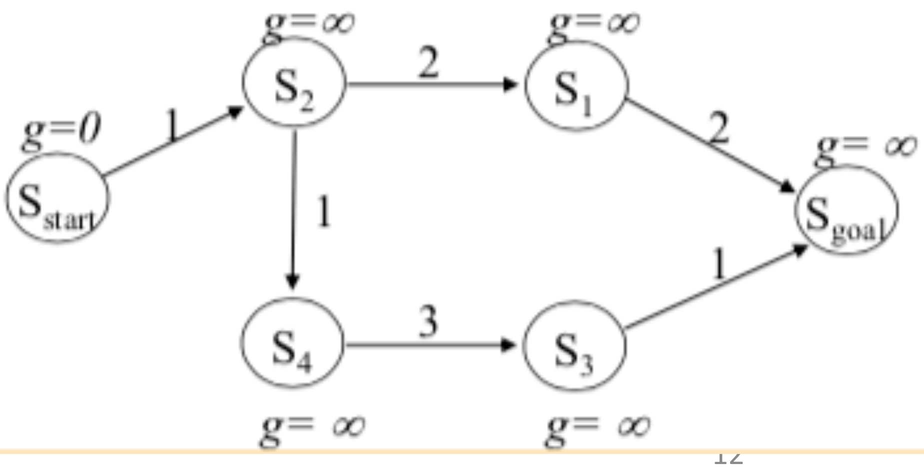
ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)
  remove s with the smallest $g(s)$ from $OPEN$;
  insert s into $CLOSED$;
  for every successor s' of s such that s' not in $CLOSED$
    if $g(s') > g(s) + c(s,s')$
      $g(s') = g(s) + c(s,s')$;
      insert s' into $OPEN$;

Check if the state is not in closed.
Decrease g*(s) if a lower-cost path is found for a state s.

set of states that have already been expanded

tries to decrease $g(s')$ using the found path from s_{start} to s



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $g(s)$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

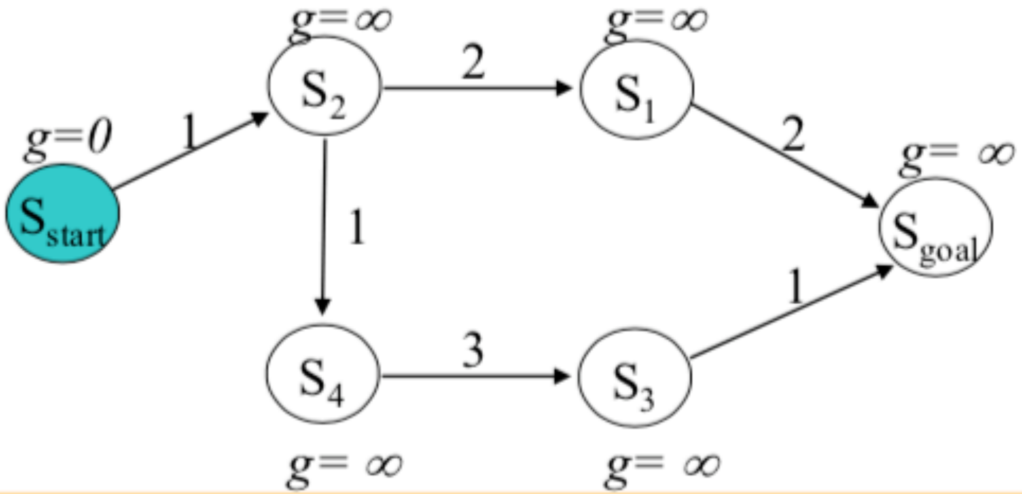
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{\}$

$OPEN = \{s_{start}\}$

next state to expand: s_{start}



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $g(s)$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

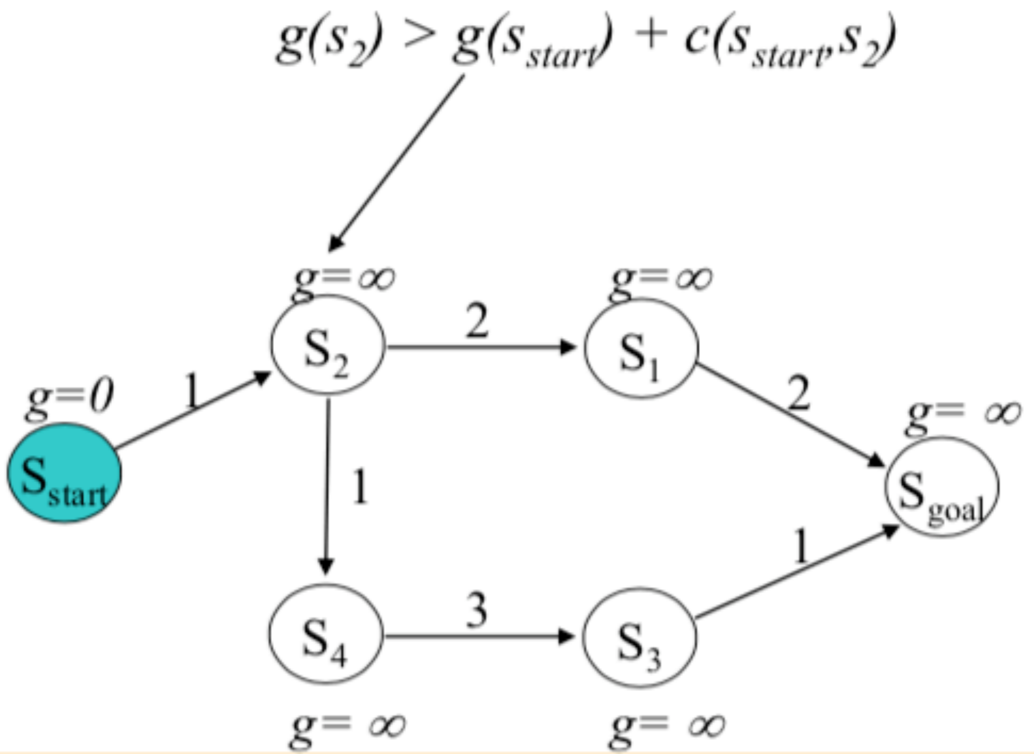
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{\}$

$OPEN = \{s_{start}\}$

next state to expand: s_{start}



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $g(s)$ from $OPEN$;

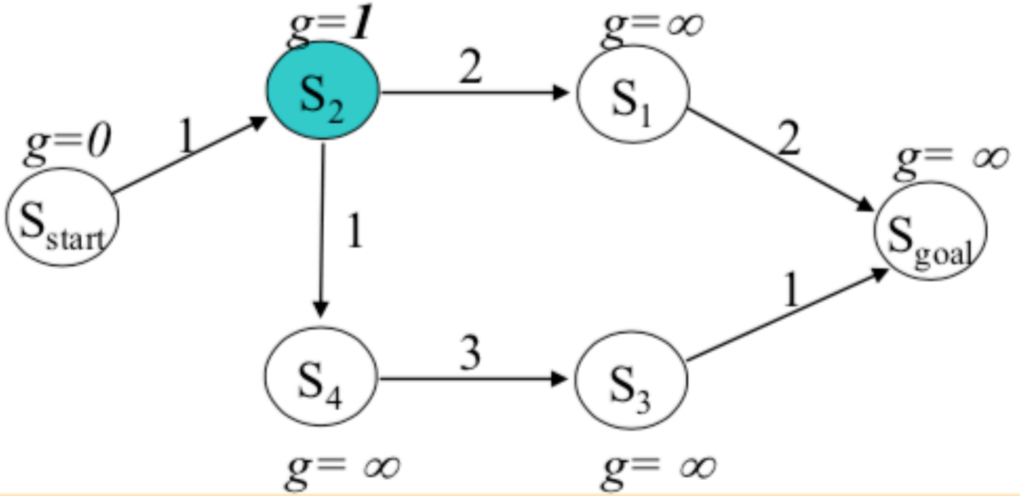
  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

      $g(s') = g(s) + c(s,s');$

      insert s' into $OPEN$;



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $g(s)$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

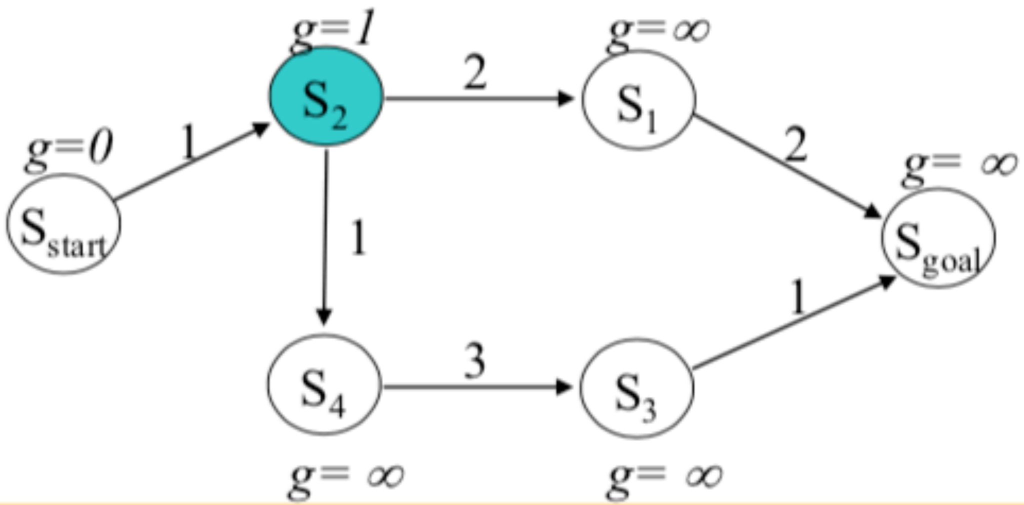
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}\}$

$OPEN = \{s_2\}$

next state to expand: s_2



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $g(s)$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

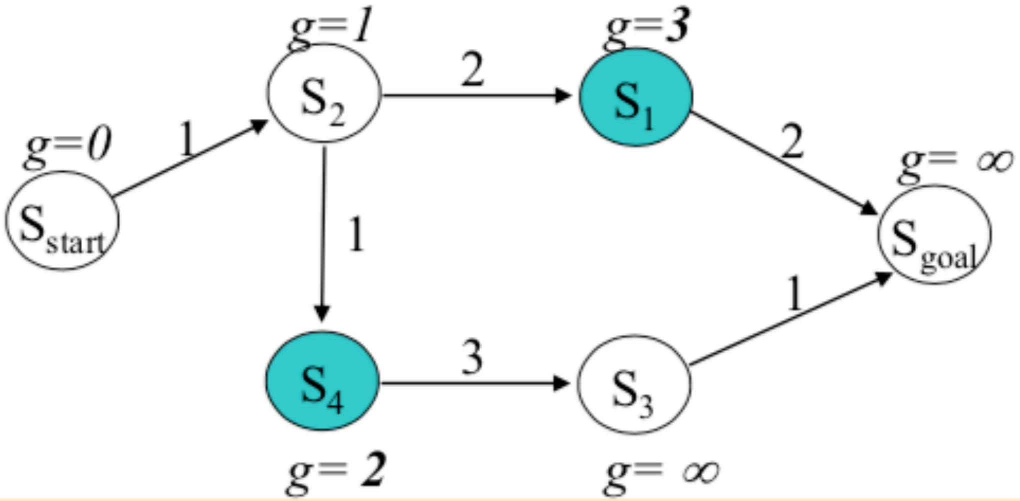
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2\}$

$OPEN = \{s_1, s_4\}$

next state to expand: ?



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $g(s)$ from $OPEN$;

  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

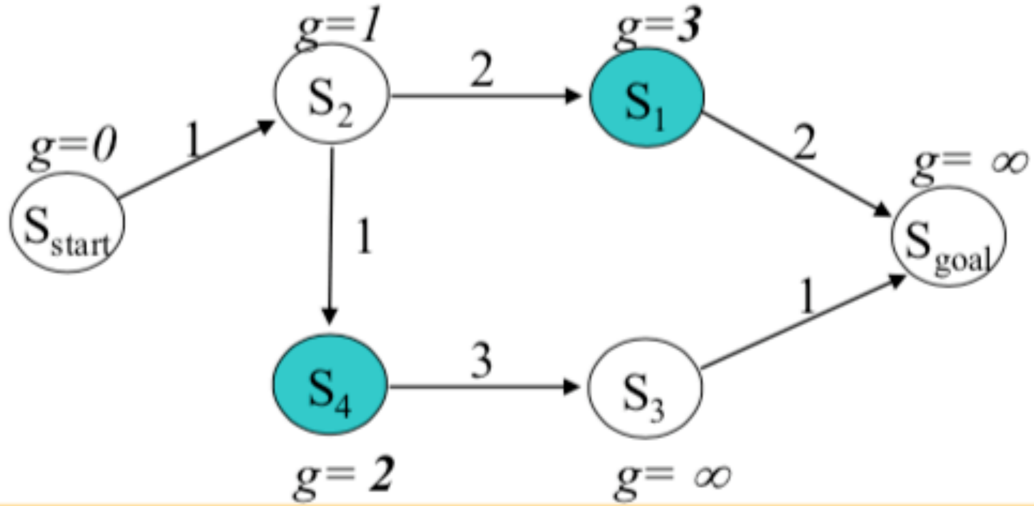
      $g(s') = g(s) + c(s,s')$;

      insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2\}$

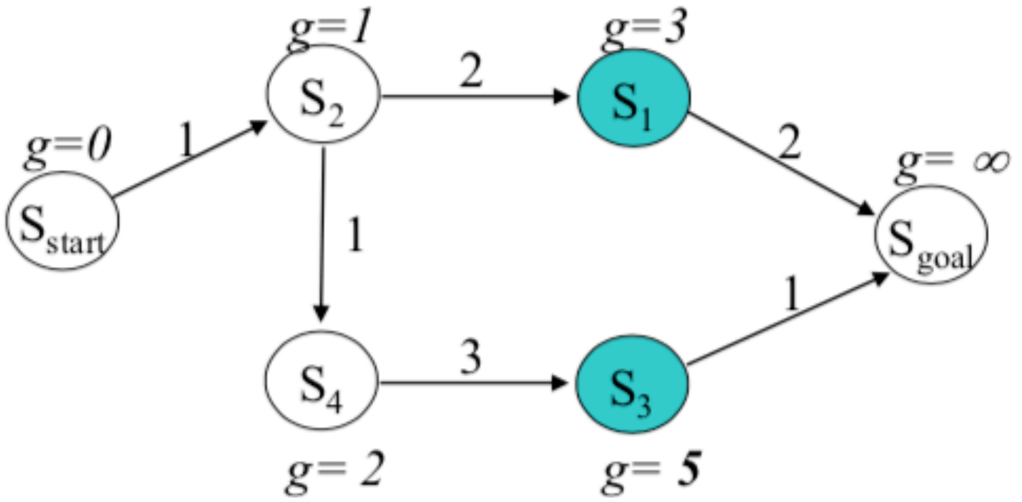
$OPEN = \{s_1, s_4\}$

next state to expand: s_4

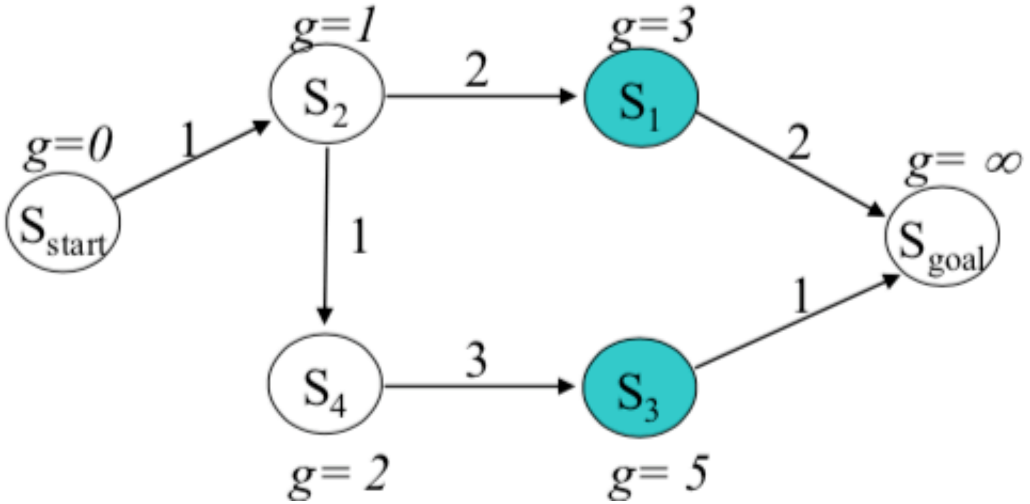


Example

CLOSED = $\{s_{start}, s_2, s_4\}$
OPEN = $\{s_1, s_3\}$
next state to expand: ?



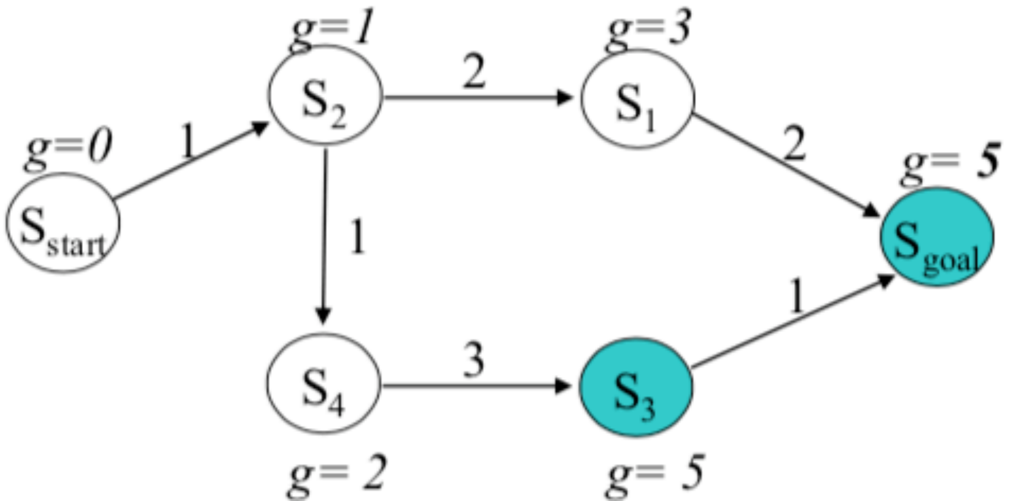
CLOSED = $\{s_{start}, s_2, s_4\}$
OPEN = $\{s_1, s_3\}$
next state to expand: s_1



Example

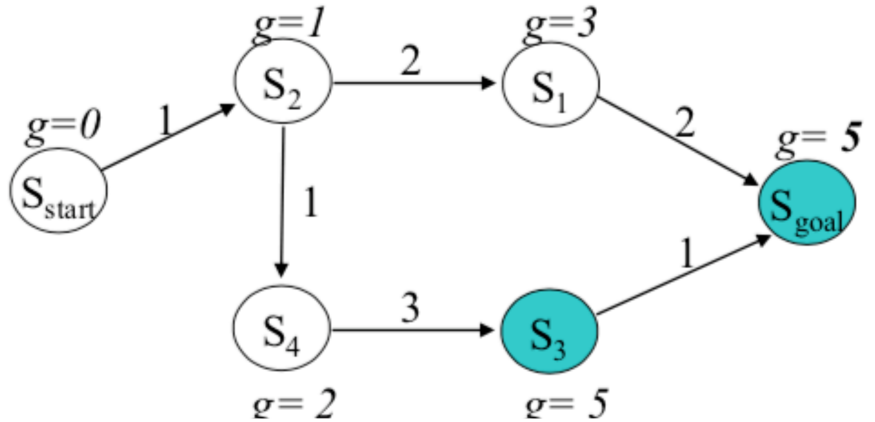
Optional optimization:
If OPEN contains multiple states with the smallest g-values and s_{goal} is one of them, then select s_{goal} for expansion (as the path through the other node will be longer).

$CLOSED = \{s_{start}, s_2, s_4, s_1\}$
$OPEN = \{s_3, s_{goal}\}$
next state to expand: ?

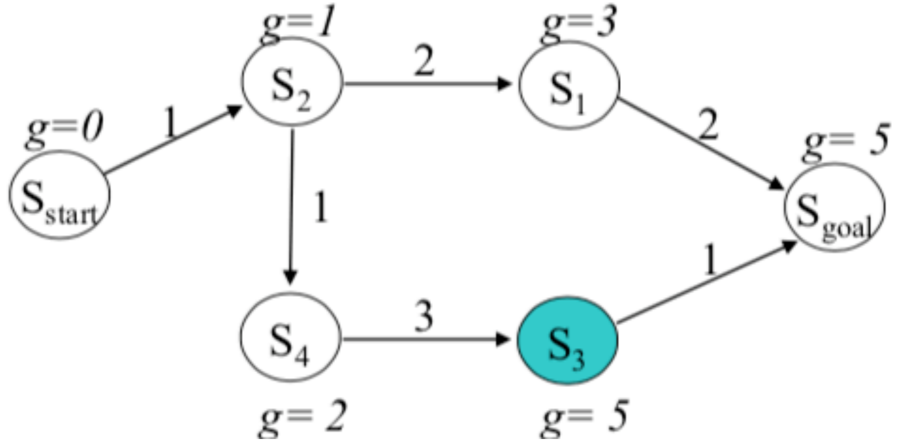


Example

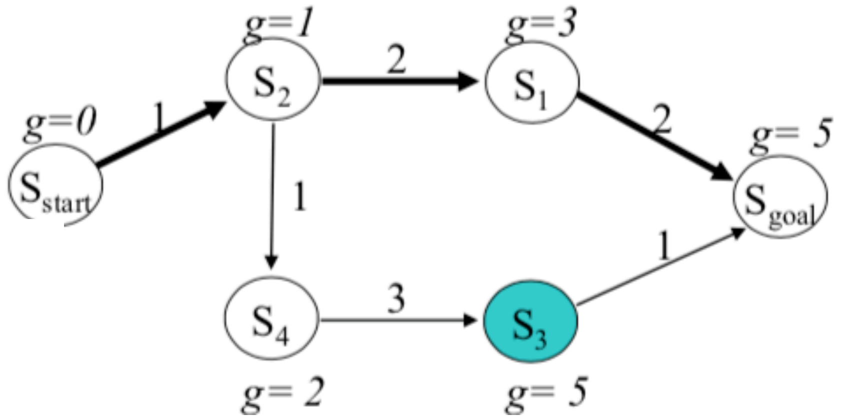
$CLOSED = \{s_{start}, s_2, s_4, s_1\}$
$OPEN = \{s_3, s_{goal}\}$
next state to expand: s_{goal}



$CLOSED = \{s_{start}, s_2, s_4, s_1, s_{goal}\}$
$OPEN = \{s_3\}$
done



Properties
- For every expanded state g(s) = g*(s)
- For every other state g(s) ≥ g*(s)
- Once the g*() values are computed, determine the least-cost path by backtracking.

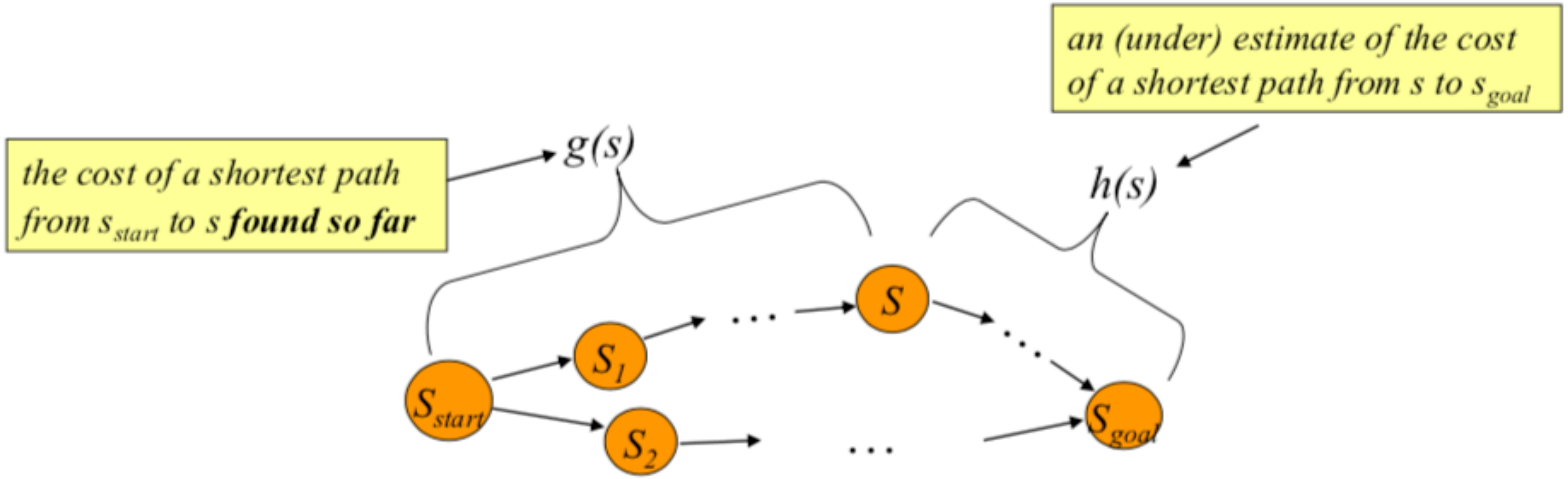


Estimating Cost-to-goal via Heuristics

- Till now we computed "cost so far"
  - The uninformed A* search expands nodes based on the cost of the node from the start node, $c(s_0, s)$
  - Till now, we are agnostic about the goal.
- While planning we often have an *intuition* about *"approximate cost to goal".*
  - If we knew the exact cost then no search would be needed.
  - But, even if we do not know $c(s, s_g)$ exactly, we often have some intuition about this distance. This intuition is called a heuristic, $h(s)$.
- Heuristic
  - h(s) = estimated cost of the **cheapest path** from the **state s to a goal state**.
  - Heuristics can be arbitrary, non-negative, **problem-specific** functions.
  - Constraint, h(s) = 0 if s is a goal.

A* Search

- Core Idea
  - Rank states by how promising they are to find the goal
  - Create a ranking by combining the "cost so far" and the "estimated cost to go".
  - Compute a function f(s) for a state that combines the two costs.
- Prioritize the exploration of nodes based on the combined ranking.
  - Always expand node with lowest f(s) first, where
    - g(s) = actual cost from the initial state to s.
    - h(s) = estimated cost from n to the next goal.
    - **f(s) = g(s) + h(s)**, the estimated cost of the cheapest solution through s. It is the cost so far and an estimate of the cost to go.



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

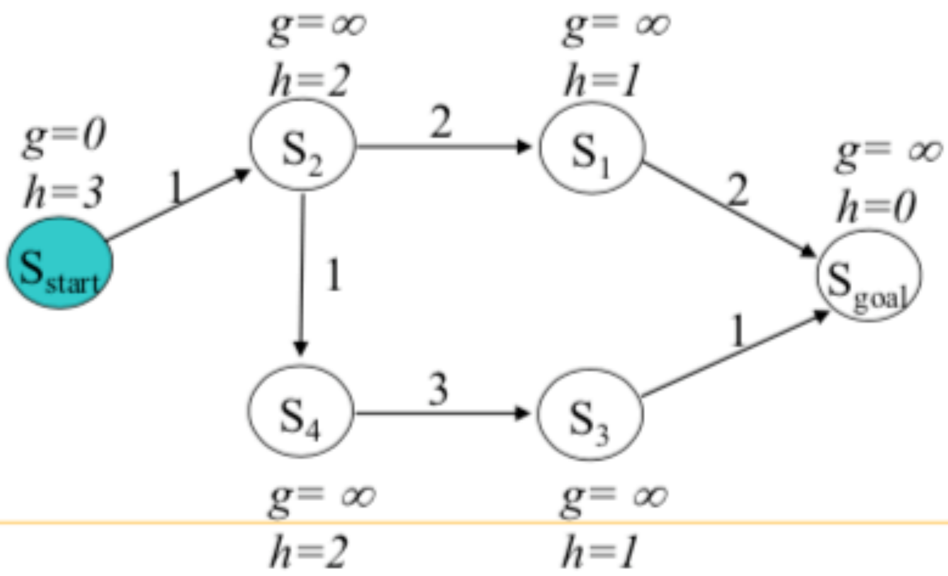
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{\}$

$OPEN = \{s_{start}\}$

next state to expand: s_{start}



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

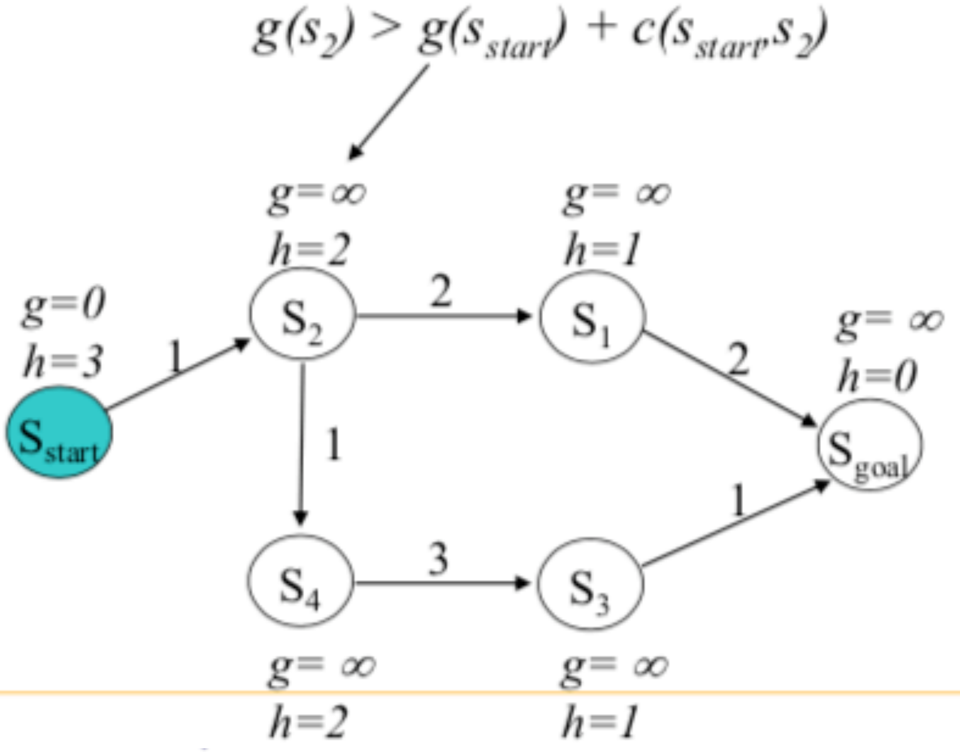
  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

      $g(s') = g(s) + c(s,s')$;

      insert s' into $OPEN$;



$CLOSED = \{\}$

$OPEN = \{s_{start}\}$

next state to expand: s_{start}

Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

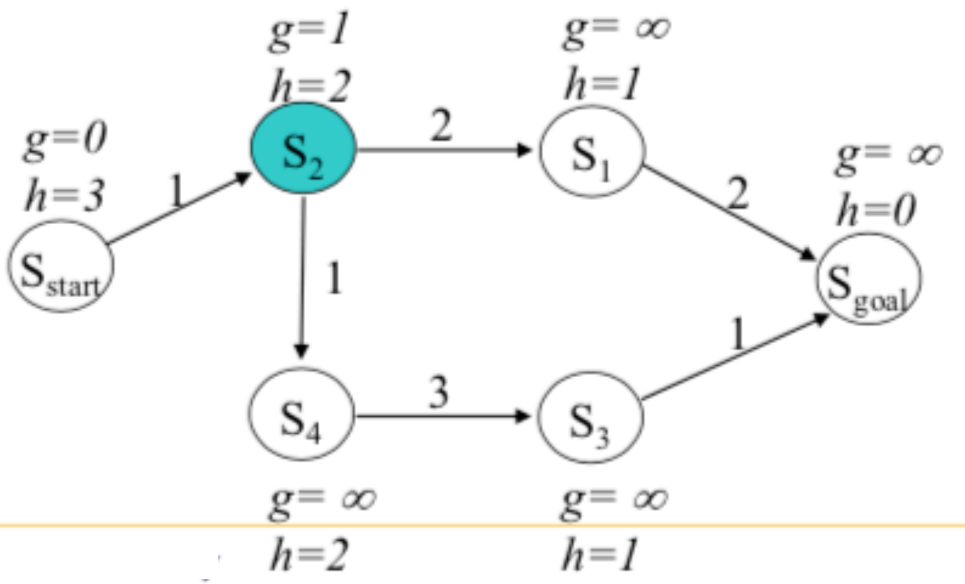
insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

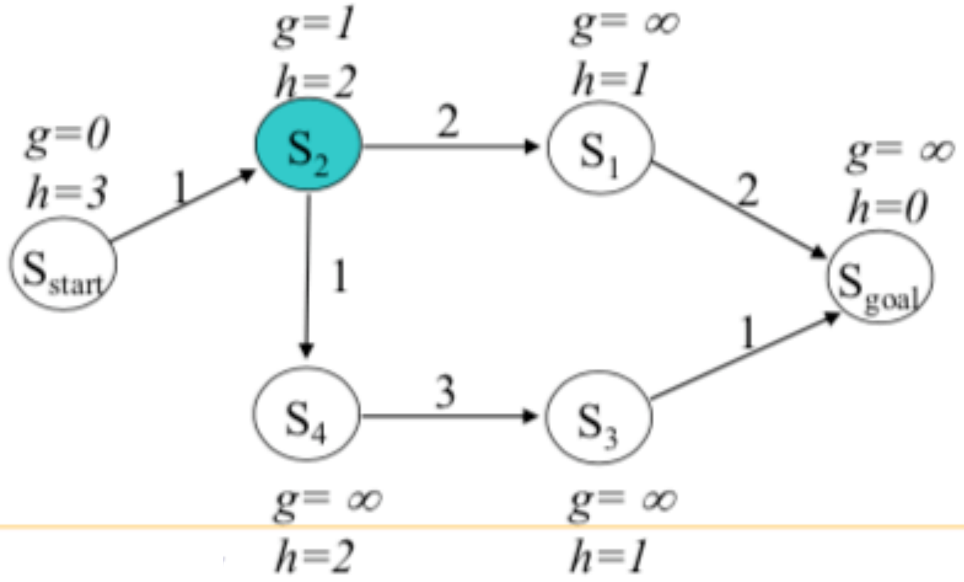
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}\}$

$OPEN = \{s_2\}$

next state to expand: s_2



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

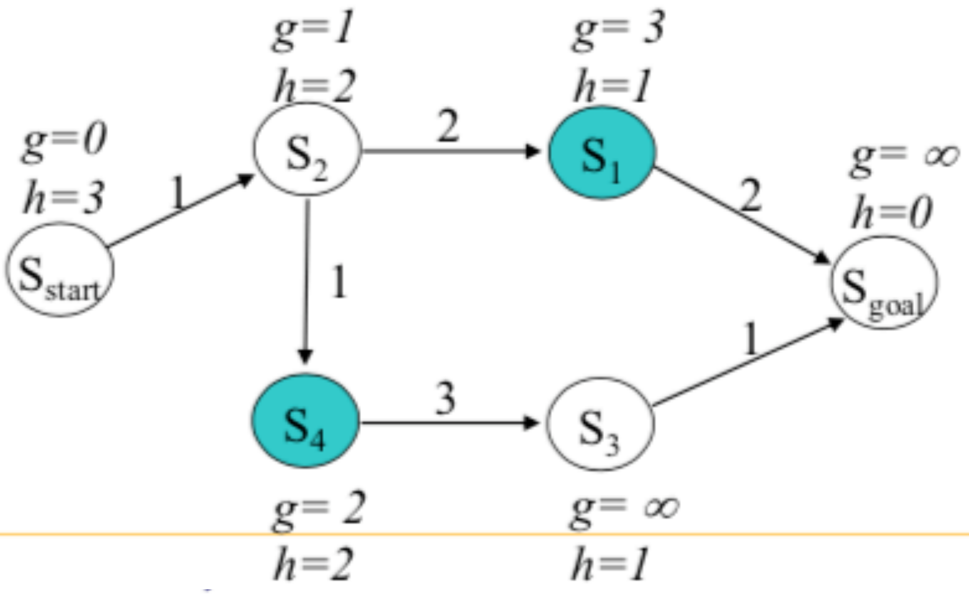
      $g(s') = g(s) + c(s,s')$;

      insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2\}$

$OPEN = \{s_1, s_4\}$

next state to expand: s_1



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

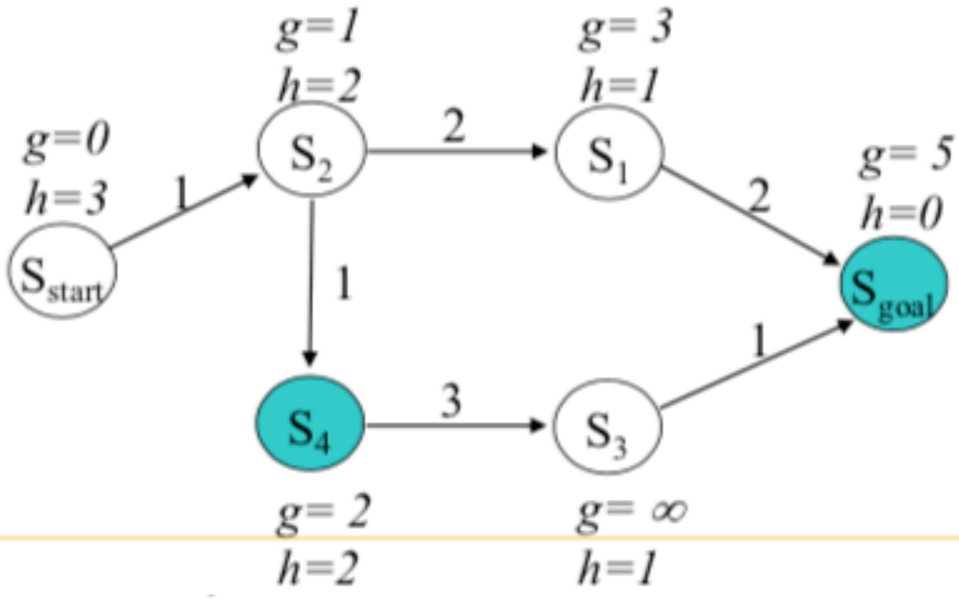
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2, s_1\}$

$OPEN = \{s_4, s_{goal}\}$

next state to expand: s_4



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

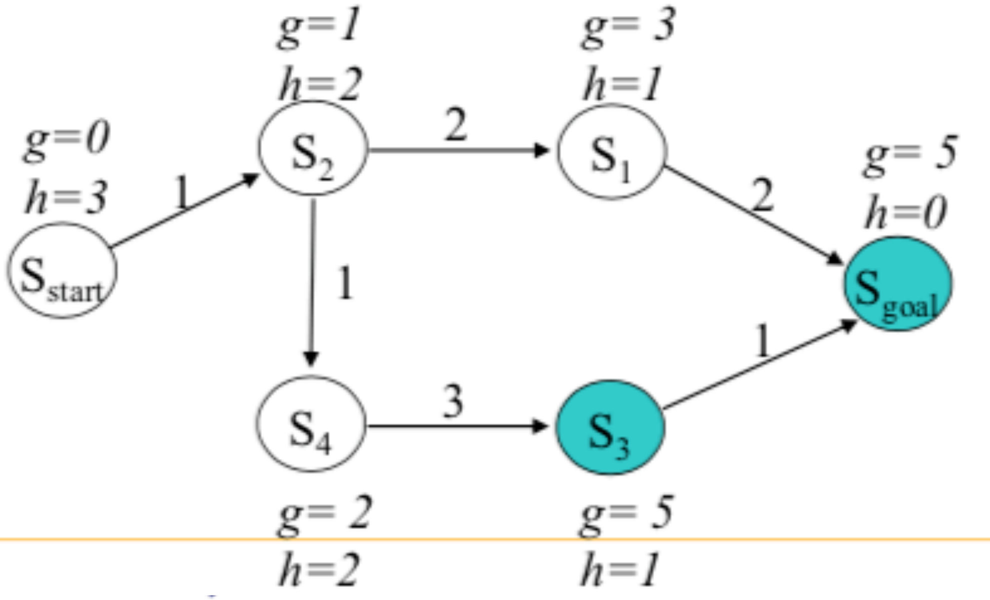
      $g(s') = g(s) + c(s,s')$;

      insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2, s_1, s_4\}$

$OPEN = \{s_3, s_{goal}\}$

next state to expand: s_{goal}



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

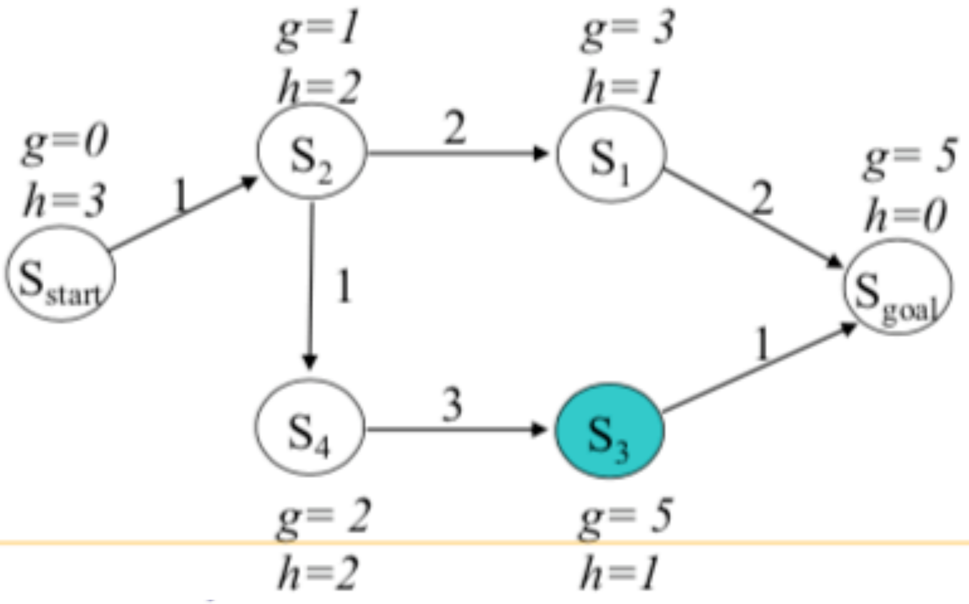
$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

$CLOSED = \{s_{start}, s_2, s_1, s_4, s_{goal}\}$

$OPEN = \{s_3\}$

done



Example

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

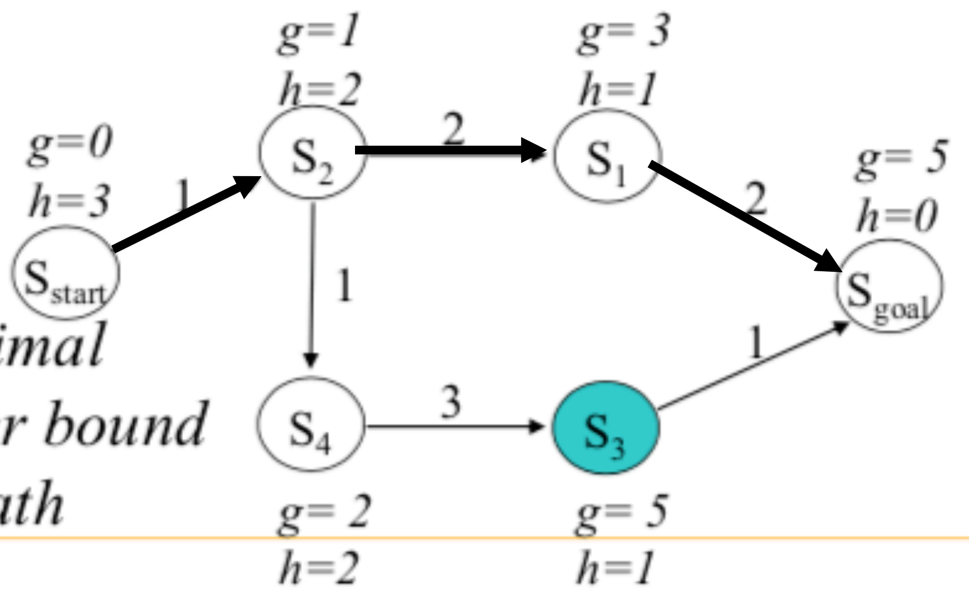
insert s into $CLOSED$;

for every successor s' of s such that s' not in $CLOSED$

if $g(s') > g(s) + c(s,s')$

$g(s') = g(s) + c(s,s')$;

insert s' into $OPEN$;

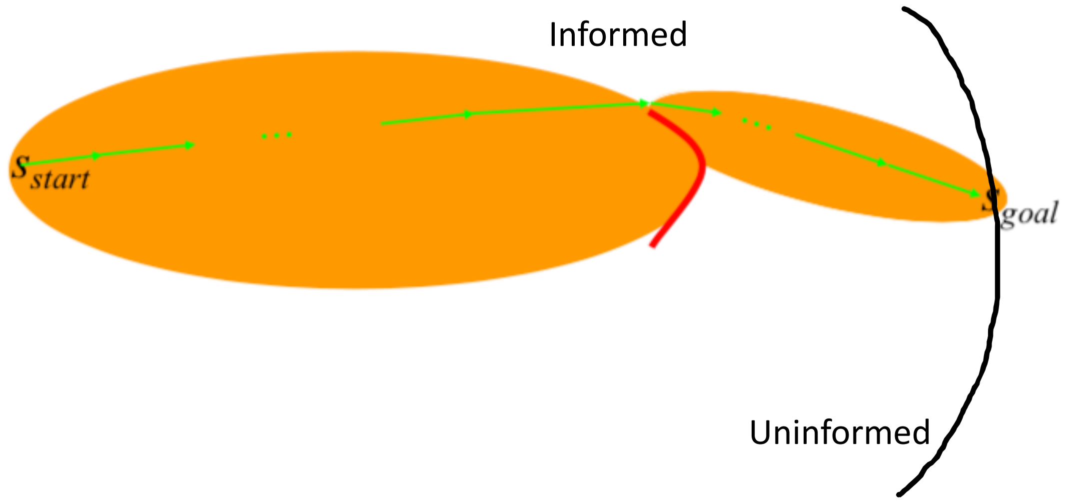


for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

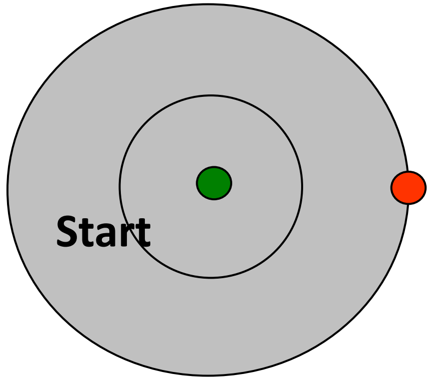
A*: Uninformed vs. Informed Search

- A*: expands states in the order of ***f* = *g*+*h*** values
- Uninformed A* or (or Uniform Cost Search) : expands states in the order of ***g*** values
- Intuitively: *f(s)* – estimate of the cost of a least cost path from start to goal via state *s*

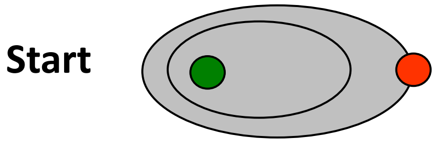
A* search with Euclidean distance heuristic.



Uninformed Search Contours



Informed Search Contours



Implementation Details

- OPEN List
  - Priority queue (common to use a binary heap)
  - Priority based on the f function.
  - Intuition
    - The queue maintains solution hypothesis.
    - Prioritization based on which states are likely to reach to the goal.
- CLOSED List
  - Typically, each state has a Boolean flag indicating that it is closed.
- Back pointers
  - After the search terminates, the least cost path is given by backtracking back pointers from s_{goal} to s_{start}

Main function

$g(s_{start}) = 0$; all other g-values are infinite; $OPEN = \{s_{start}\}$;

set all backpointers bp to NULL;

ComputePath();

publish solution; //**backtrack least-cost path using backpointers *bp***

ComputePath function

while(s_{goal} is not expanded and $OPEN \neq 0$)

  remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;

  insert s into $CLOSED$;

  for every successor s' of s such that s' not in $CLOSED$

    if $g(s') > g(s) + c(s,s')$

      $g(s') = g(s) + c(s,s')$; **bp(s') = s;**

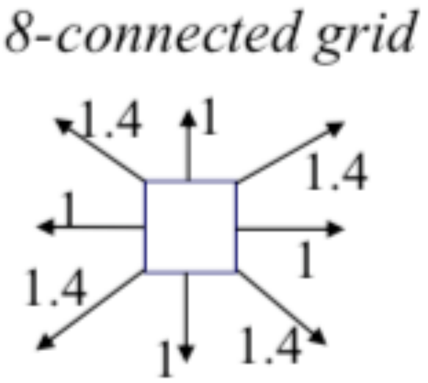
      insert s' into $OPEN$;

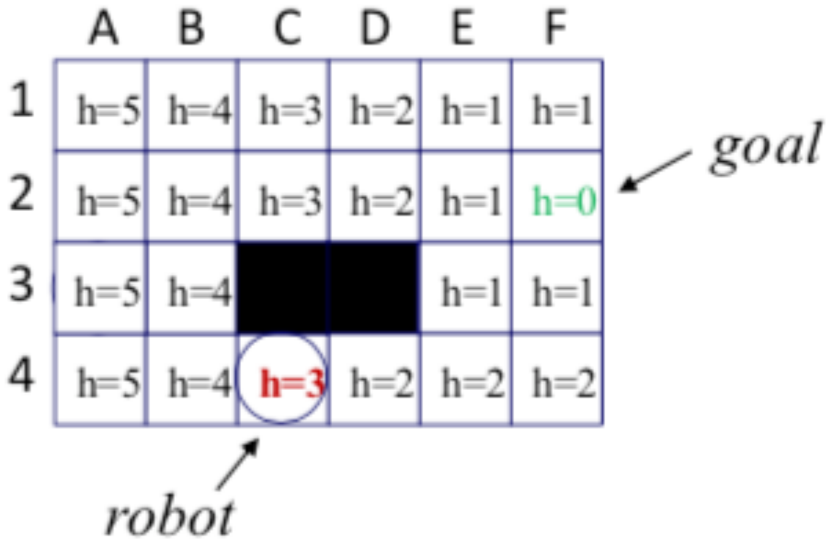
When the min-cost path is updated, also update the back pointer.

Example

- A heuristic for a grid-based graph

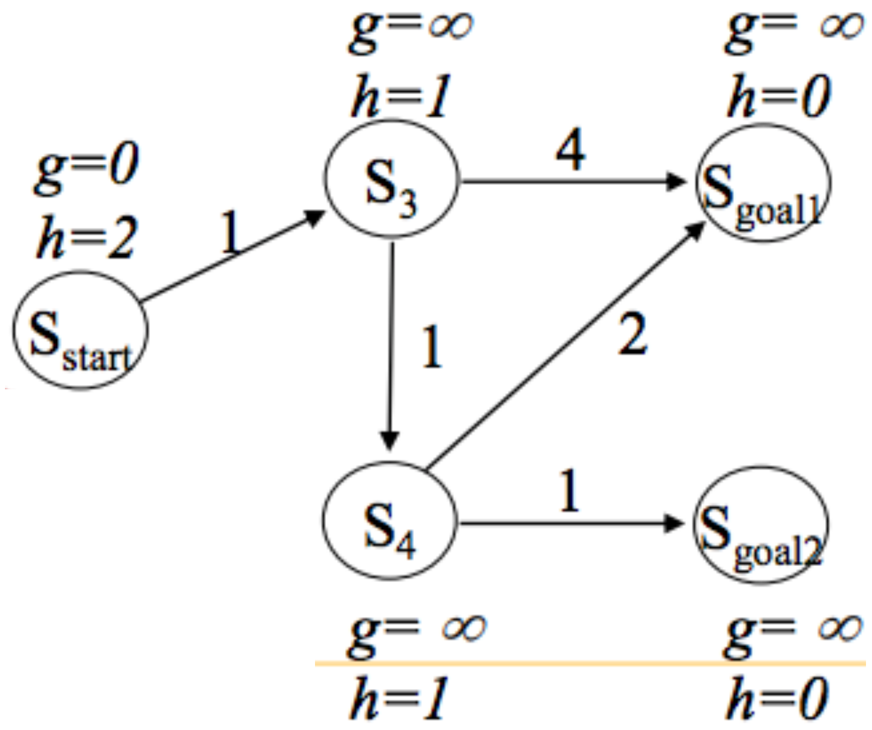
$$h(cell\ \langle x,y \rangle) = max(|x-x_{goal}|, |y-y_{goal}|)$$





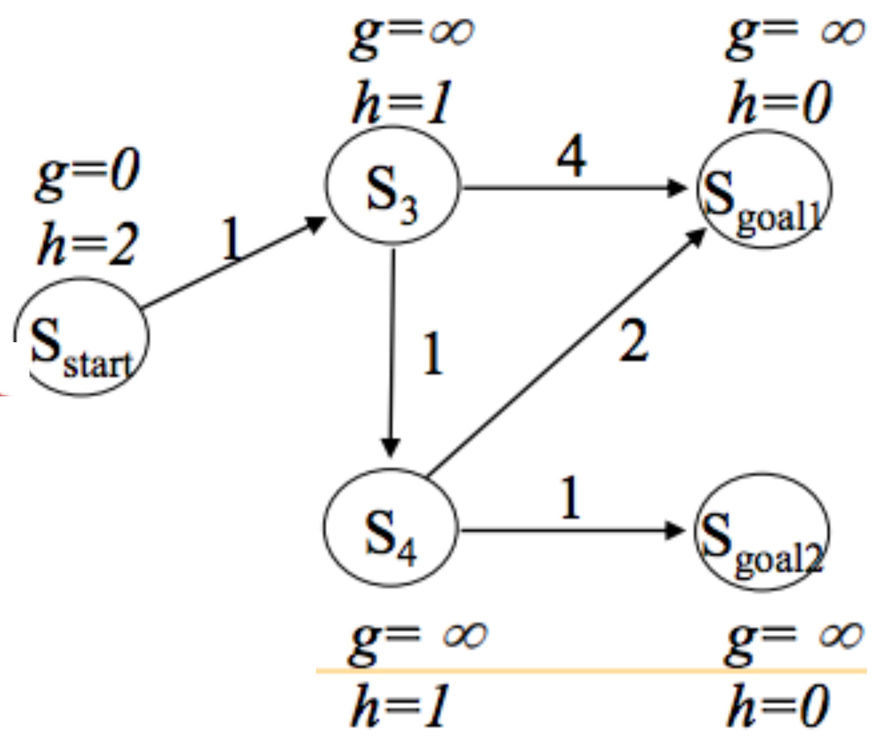
Support for Multiple Goal Candidates

- Examples
  - A robot is to reach a parking location.
    - Choice of locations some are closer, and some are further away.
  - The agent wants to escape from a room and there are multiple exits.
    - Can only escape via a door.
- How to plan in the presence of multiple goals?
  - How to find a least cost path that is lowest across all possible goals?

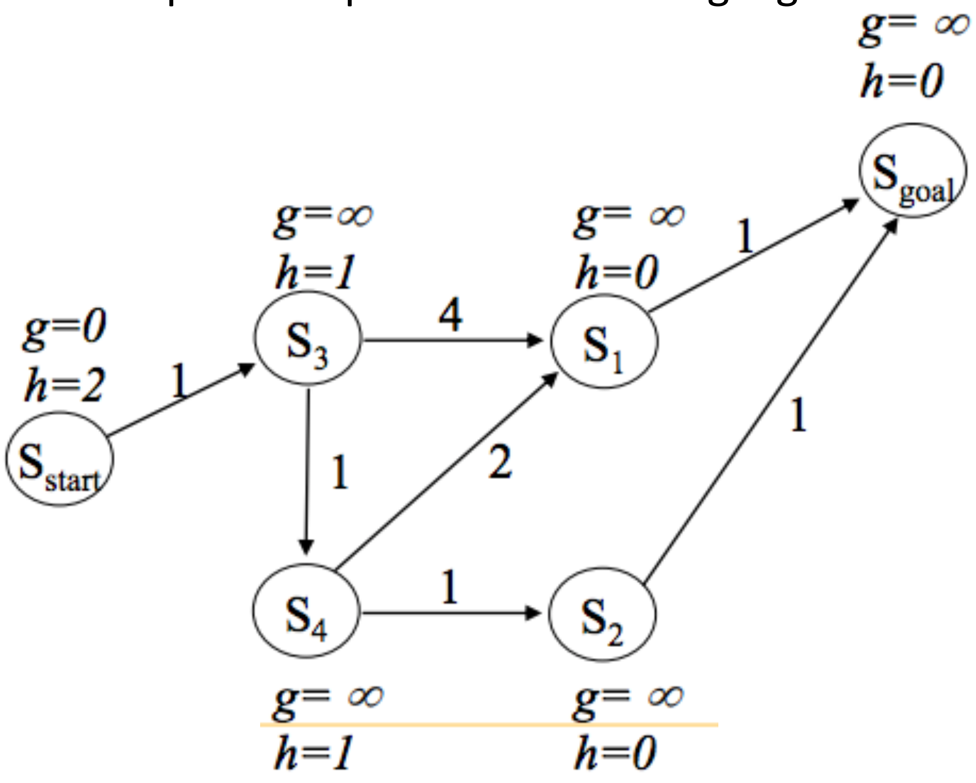


Multi-Goal A*: Introducing “Imaginary” Goal

Multiple-goal problem

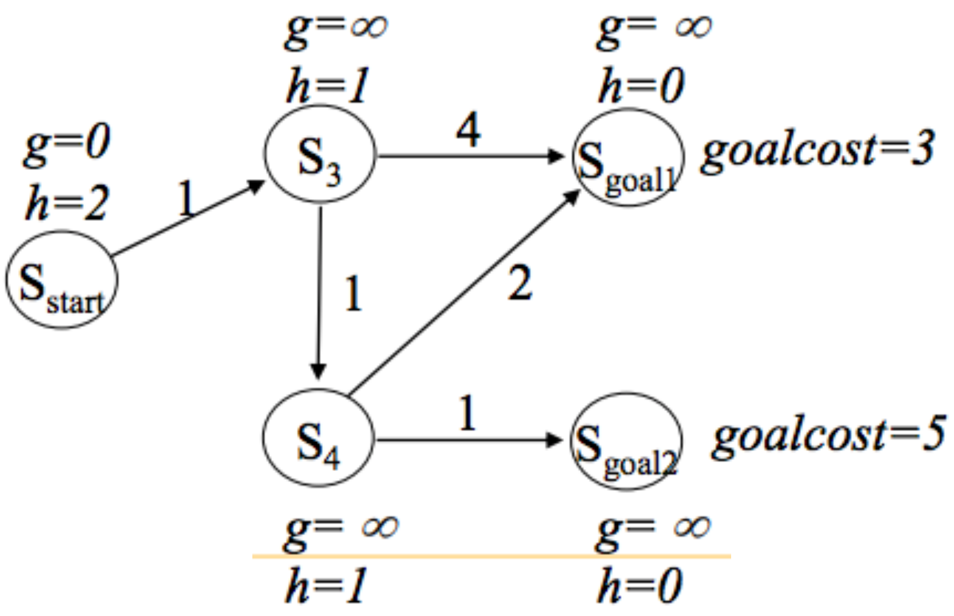


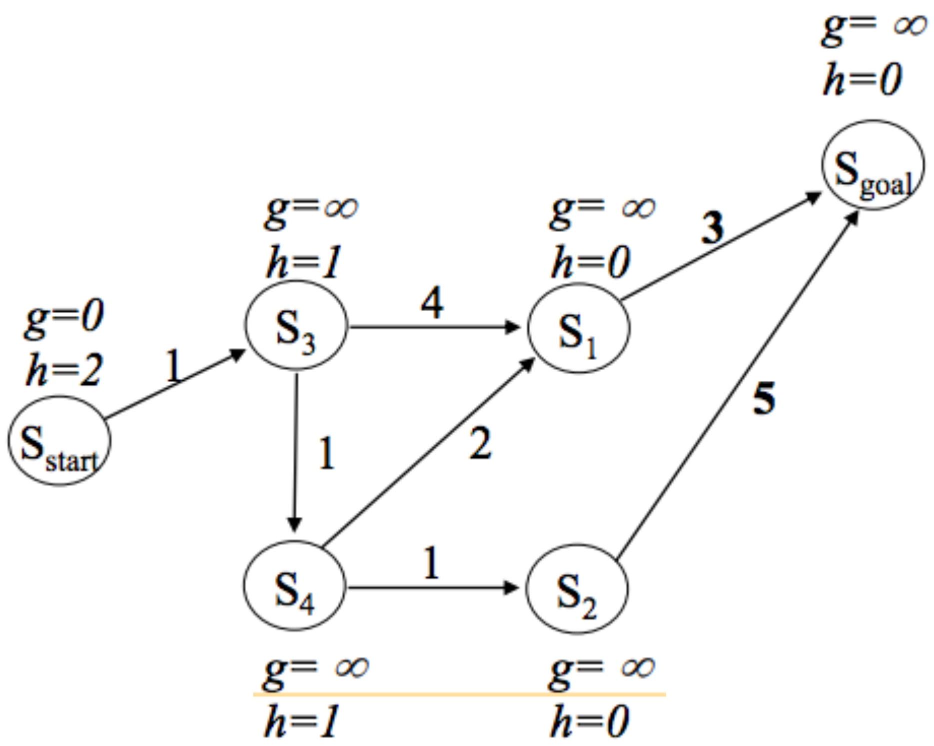
Equivalent problem with a single goal



Transform the graph with an “imaginary goal”. Following which run A*. The augmentation helps pick one goal from the many goals.

Multi-Goal A*: What if some goals are better?





The non-uniform goal preferences can be encoded as edge costs.

Heuristics: Admissibility and Consistency

- **Admissibility**
  - Let **h*(n)** be the shortest path from n to any goal state.
  - Heuristic h is called *admissible* if **$h(n) \leq h^*(n)\ \forall n$**.
  - Admissible heuristics are *optimistic*, they often think that the cost to the goal is less than actual
  - If h is admissible, then $h(g) = 0,\ \forall g \in G$
  - A trivial case of an admissible heuristic is $h(n) = 0,\ \forall n$.
- **Consistency (monotonicity)**
  - An admissible heuristic h is called consistent if for every state s and for every successor s’, **$h(s) \leq c(s, s') + h(s')$**
  - This is a version of triangle inequality, so heuristics that respect this inequality are metrics.
  - Consistency is a stricter requirement than admissible. If consistent then the heuristic is admissible.

Heuristics: Dominance

- **Dominance**
  - Comparing two heuristics.
  - Heuristic function h_2 (strictly) dominates h_1 if
    - both are admissible and
    - for every node n, $h_2(n)$ is (strictly) greater than $h_1(n)$.
  - What is the implication?
    - A* search with a dominating heuristic function h2 will never expand more nodes that A* with h1.
    - Expansion of fewer nodes implies efficiency gains.

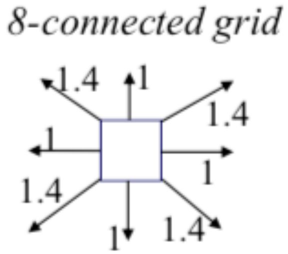
A* Search Properties

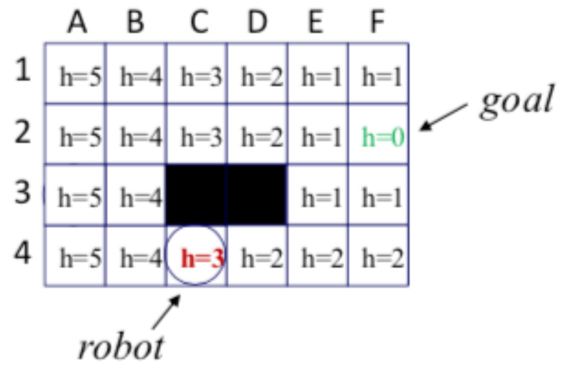
- We covered the “graph-search” version of A* in this lecture.
  - I.e., we maintain a CLOSED list.
- Optimal
  - If the heuristic is *consistent* (stronger condition than admissibility) then A* search (graph search version) will find the optimal solution.
- Completeness
  - If a solution exists, then A* will find it (eventually A* will visit all nodes)
  - Under some conditions
    - Every node has a finite number of successor nodes (b is finite). Number of nodes is finite.
    - Positive costs for edges.

Admissible Heuristics from Relaxed Problems

- Optimal solution in the original problem is also a solution for the relaxed problem.
- Cost of the optimal solution in the relaxed problem is an admissible heuristic in the original problem.
  - At least this much work is to be done during search.
- Finding the optimal solution in the relaxed problem should be “easy”
  - Without performing search.

$$h(cell\ <x,y>) = max(|x-x_{goal}|, |y-y_{goal}|)$$

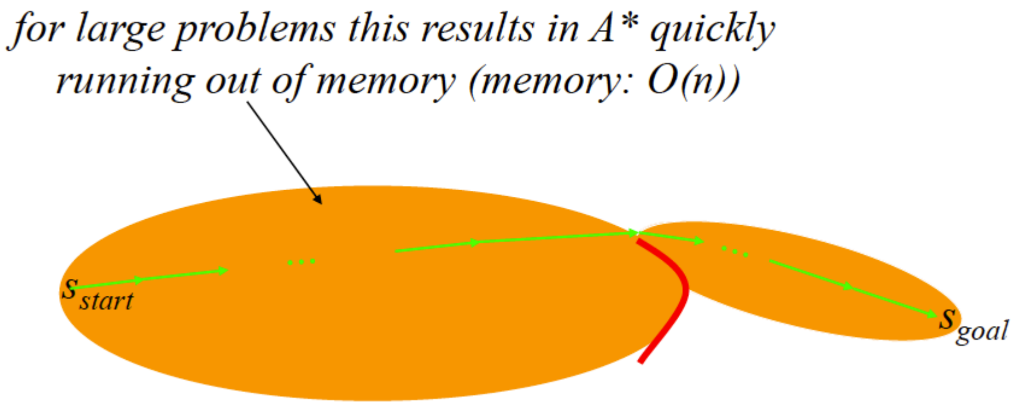




A* Search: Finding sub-optimal solutions

- Problem with A* search
  - Despite the heuristic, the priority queue can be very large.
    - A* takes too long to find the optimal solution, memory runs out.
    - Note that A* will give the optimal solution.
- Can we do fewer expansions?
  - Trading off optimality.
- In essence, how can we modify A* such that sub-optimal solutions can be found *quickly*?

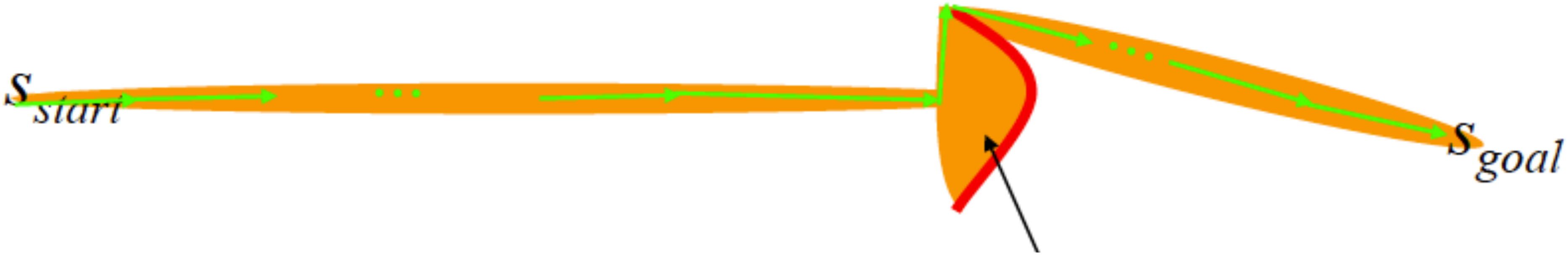
Problem with A*



Weighted A*

- Modify the prioritization function
  - Expands states in the order of ***f'(n) = g(n) + w*h(n)*** values, where ***w > 1.0***

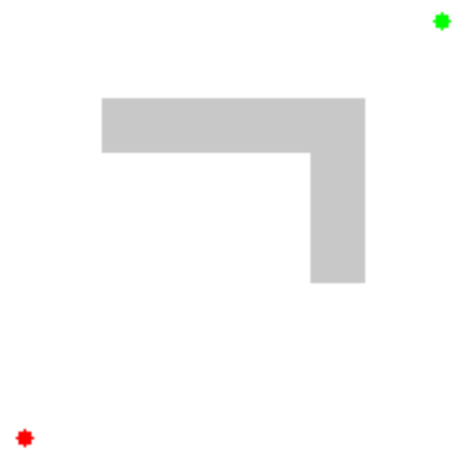
A weighted heuristic accelerates the search by making nodes closer to the goal more attractive, the cost to goal starts to dominate.



Weighted A*

- What is the effect?
    - Creates a bias towards expansion of states that are closer to goal.
    - f’(n) is *not admissible* but finds good *sub-optimal* solutions *quickly*.
    - Trade off between search effort and solution quality.
    - Usually, orders of magnitude faster than A*.

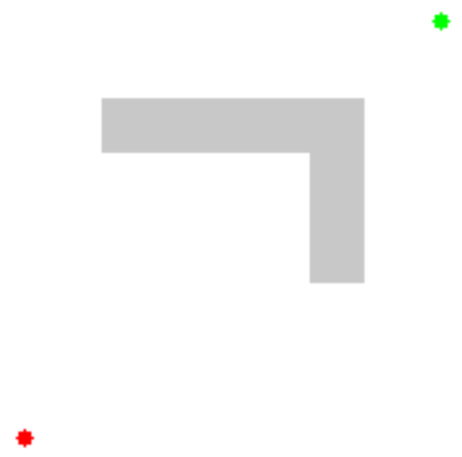
Effect of running towards the goal. May lead to sub-optimality.



Weighted A*

- What is the effect?
  - Creates a bias towards expansion of states that are closer to goal.
  - f’(n) is *not admissible* but finds good *sub-optimal* solutions *quickly*.
  - Trade off between search effort and solution quality.
  - Usually, orders of magnitude faster than A*.

Effect of running towards the goal. May lead to sub-optimality.



Planning during Execution

- One off plans may not work.
- May need to repeat the process
  - Various kinds of errors
    - Imperfect plan execution.
      - Did not land up at the right grid cell.
    - Something in the environment is now visible or changed.
      - A door is now closed.
- How to replan fast?
  - Anytime heuristic search
    - Return the best plan possible within T msecs
  - If you have more time, you can improve the plan.

Anytime Planning with weighted A*

Constructing anytime search based on weighted A*:
- Find the best path possible given some amount of time for planning
- Run a series of weighted A* searches with decreasing ε *(the weight w in the last slides):*