A and Weighted A* Search*

Maxim Likhachev
Carnegie Mellon University

Planning as Graph Search Problem

1. Construct a graph representing the planning problem

2. Search the graph for a (hopefully, close-to-optimal) path

The two steps above are often interleaved

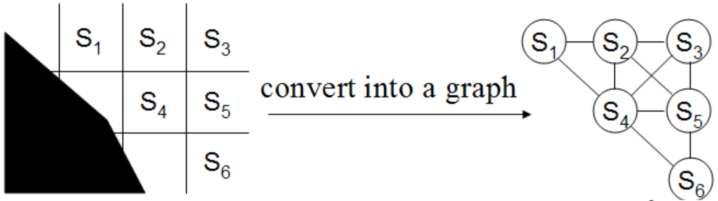
Planning as Graph Search Problem

1. Construct a graph representing the planning problem **(future lectures)**
2. Search the graph for a (hopefully, close-to-optimal) path **(three next lectures)**

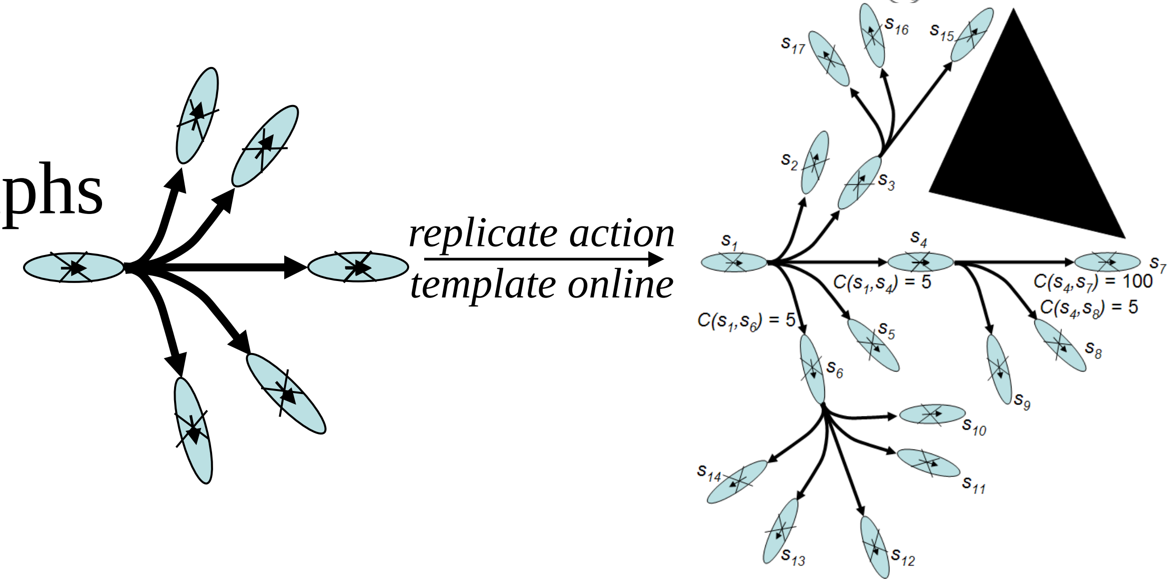
The two steps above are often interleaved

Examples of Graph Construction

- Cell decomposition
    - X-connected grids



    - lattice-based graphs

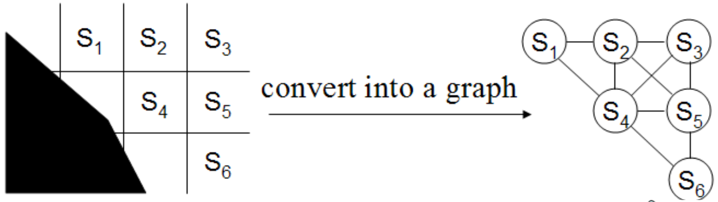


- Skeletonization of the environment/C-Space
    - Visibility graphs
    - Voronoi diagrams
    - Probabilistic roadmaps

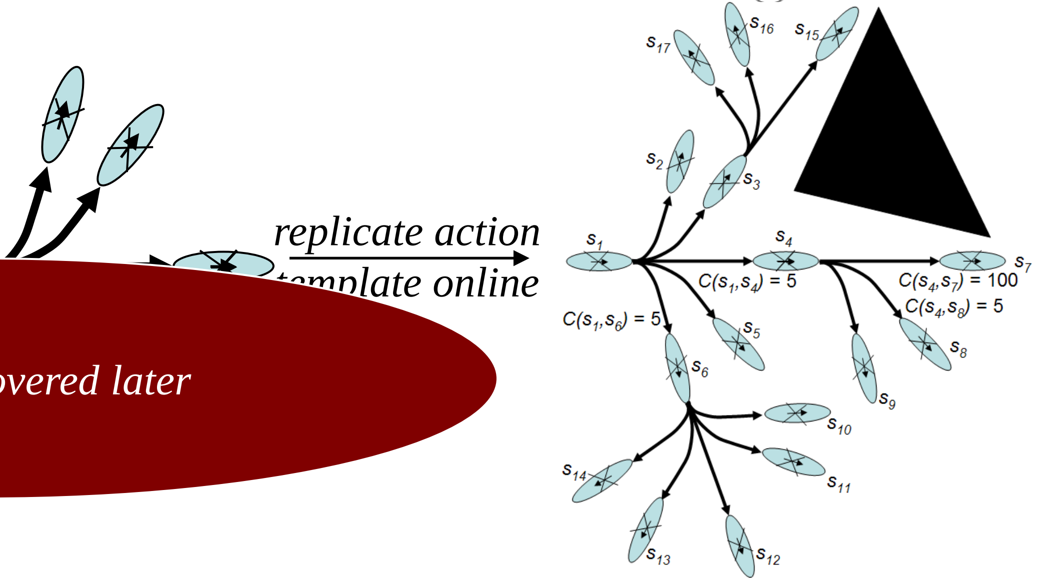
Examples of Graph Construction

- Cell decomposition

  - X-connected grids



  - lattice-based graphs



Will all be covered later

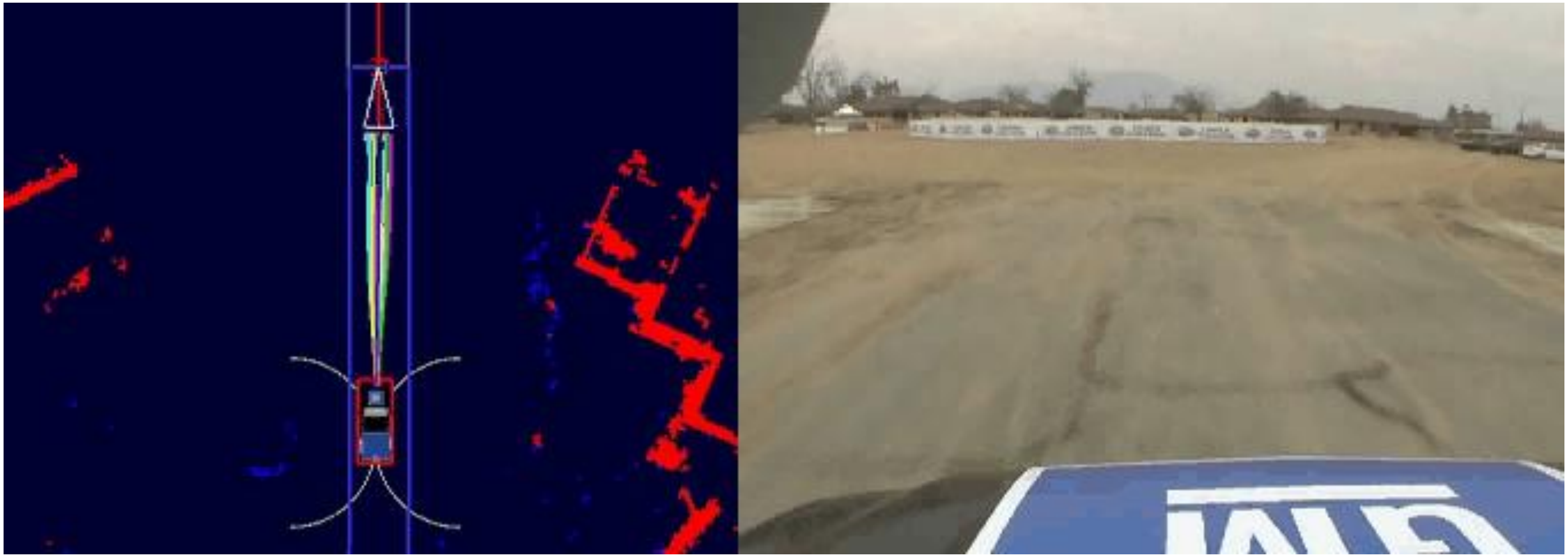
- Skeletonization of the environment/C-Space
  - Visibility graphs
  - Voronoi diagrams
  - Probabilistic roadmaps

Examples of Search-based Planning

1. *Construct a graph representing the planning problem*
2. *Search the graph for a (hopefully, close-to-optimal) path*

The two steps are often interleaved

motion planning for autonomous vehicles in 4D (<x,y,orientation,velocity>)
running Anytime Incremental A* (Anytime D*) on multi-resolution lattice
[Likhachev & Ferguson, IJRR'09]



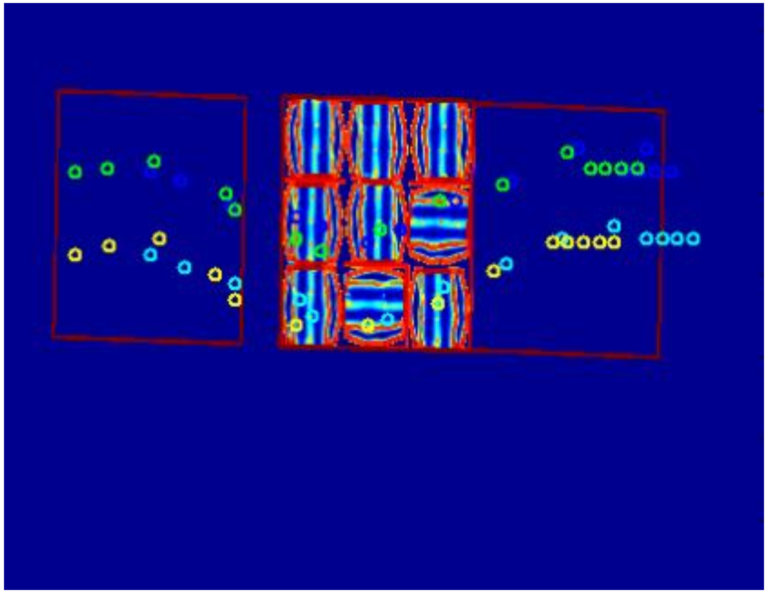
part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race

Examples of Search-based Planning

1. *Construct a graph representing the planning problem*
2. *Search the graph for a (hopefully, close-to-optimal) path*

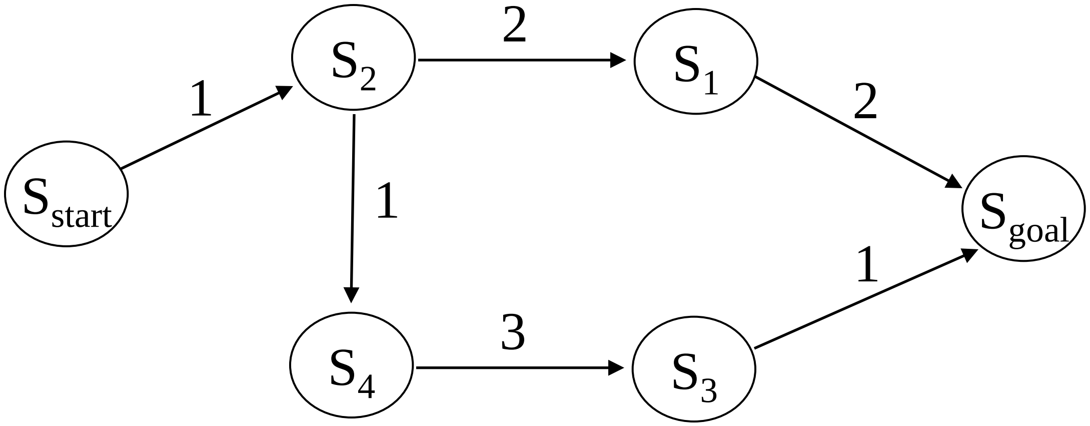
The two steps are often interleaved

8-dim foothold planning for quadrupeds using R* graph search



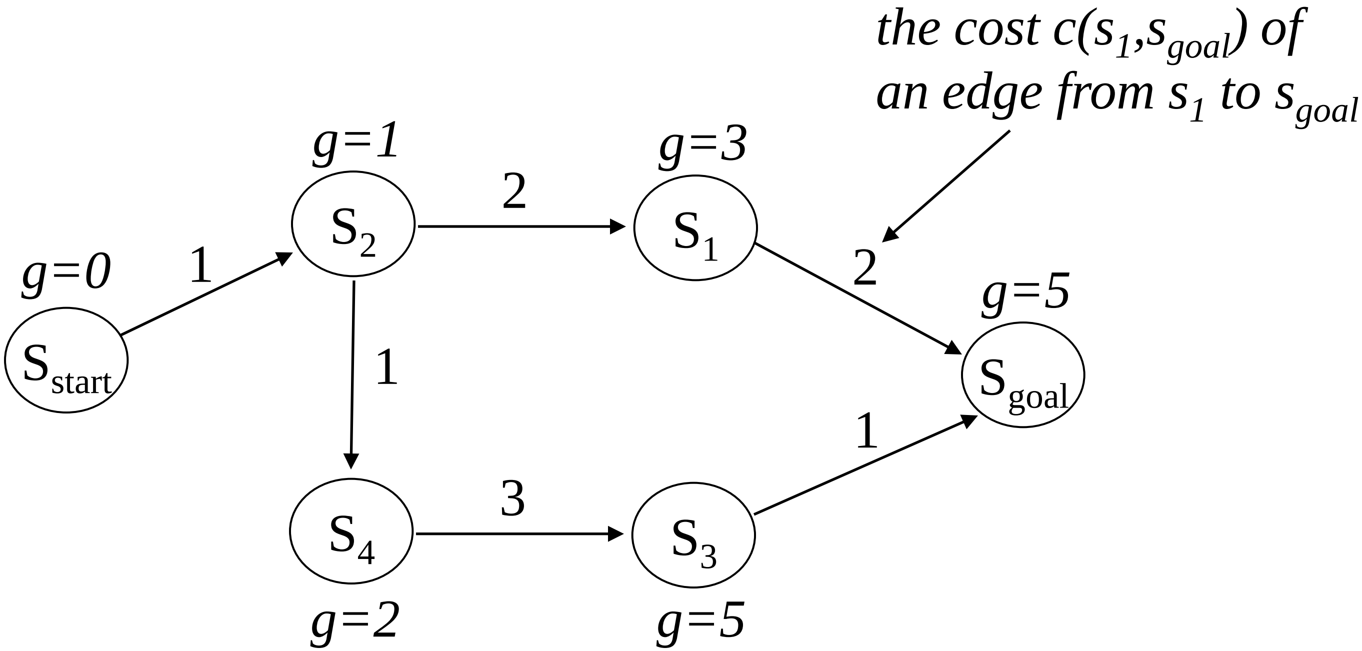
Searching Graphs for a Least-cost Path

- Once a graph is constructed (from skeletonization or uniform cell decomposition or adaptive cell decomposition or lattice or whatever else), we need to search it for a least-cost path



Searching Graphs for a Least-cost Path

- Many searches work by computing optimal g-values for relevant states
  - $g(s)$ – an estimate of the cost of a least-cost path from s_{start} to s
  - optimal values satisfy: $g(s) = \min_{s'' \in pred(s)} g(s'') + c(s'',s)$



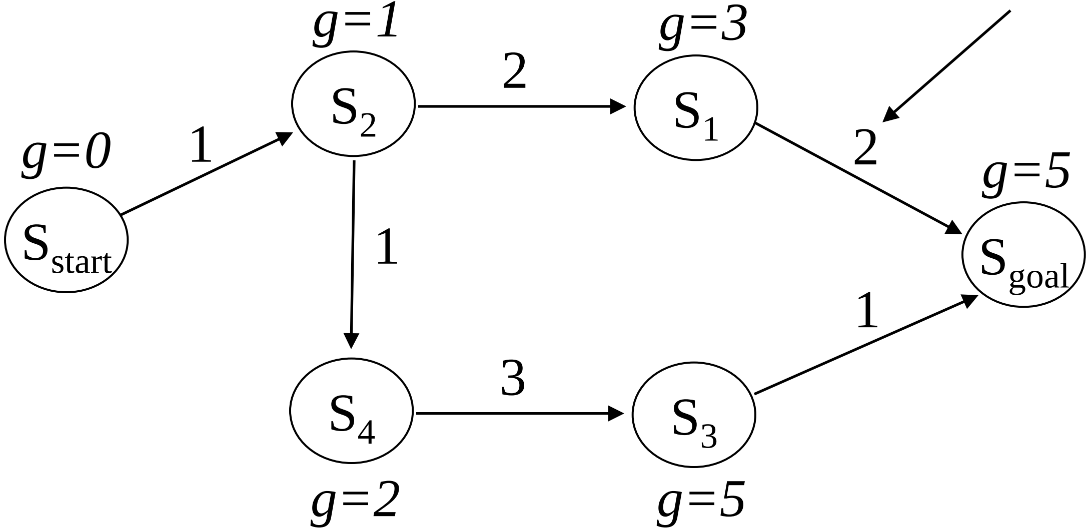
Searching Graphs for a Least-cost Path

- Many searches work by computing optimal g-values for relevant states
  - $g(s)$ – an estimate of the cost of a least-cost path from s_{start} to s
  - optimal values satisfy: $g(s) = \min_{s'' \in pred(s)} g(s'') + c(s'',s)$

the cost $c(s_1,s_{go}$

an edge from s_1

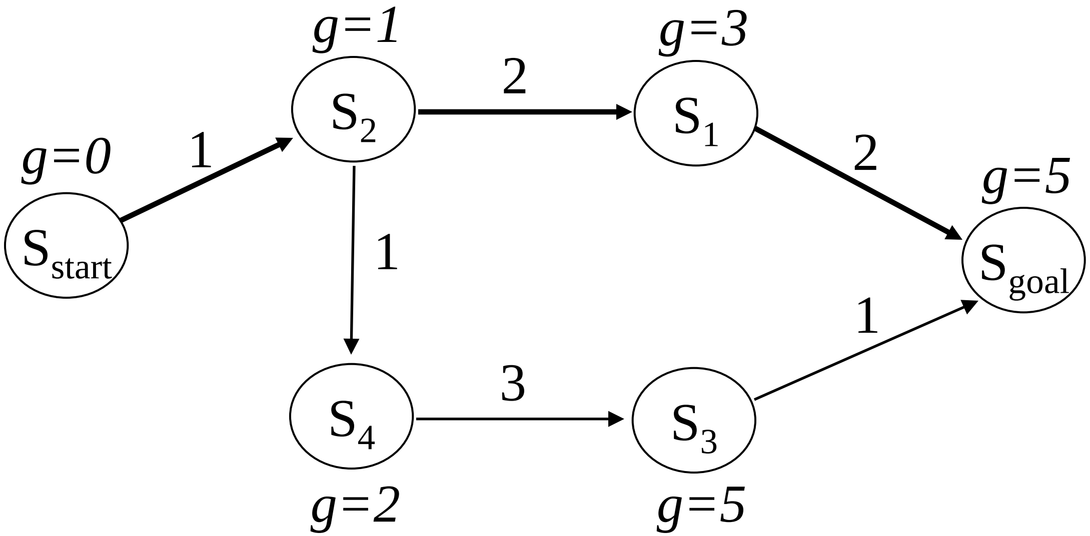
why?



Searching Graphs for a Least-cost Path

- Least-cost path is a greedy path computed by backtracking:
  - start with s_{goal} and from any state s move to the predecessor state s' such that

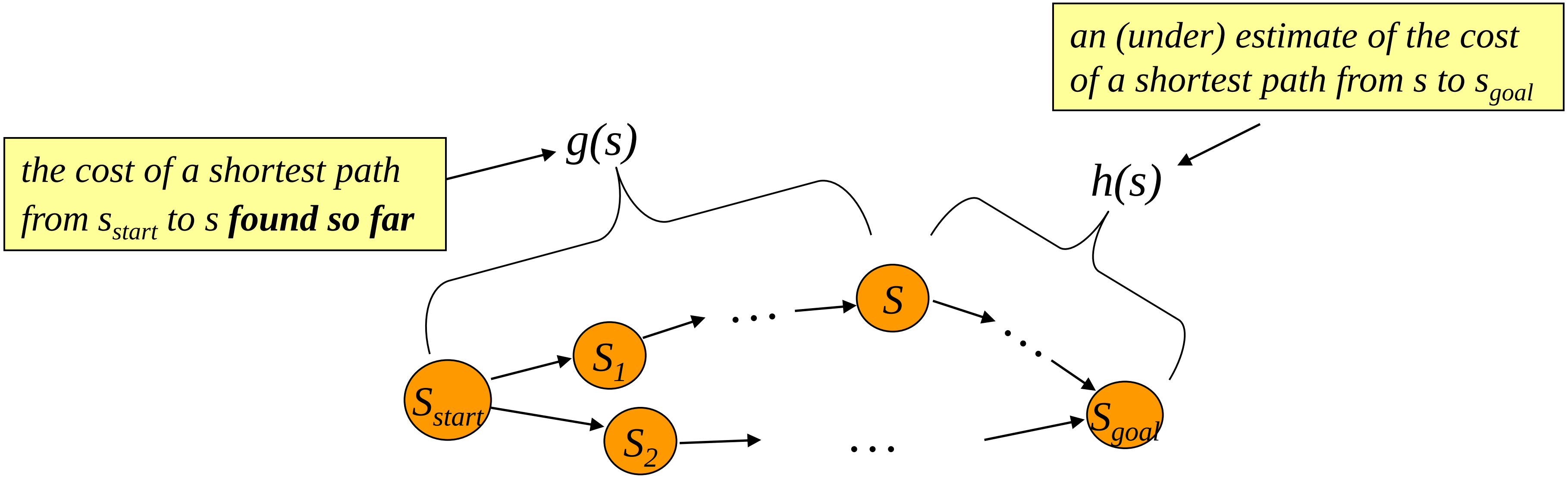
$$s' = \arg\min_{s'' \in pred(s)} (g(s'') + c(s'', s))$$



A* Search

- Computes optimal g-values for relevant states

at any point of time:



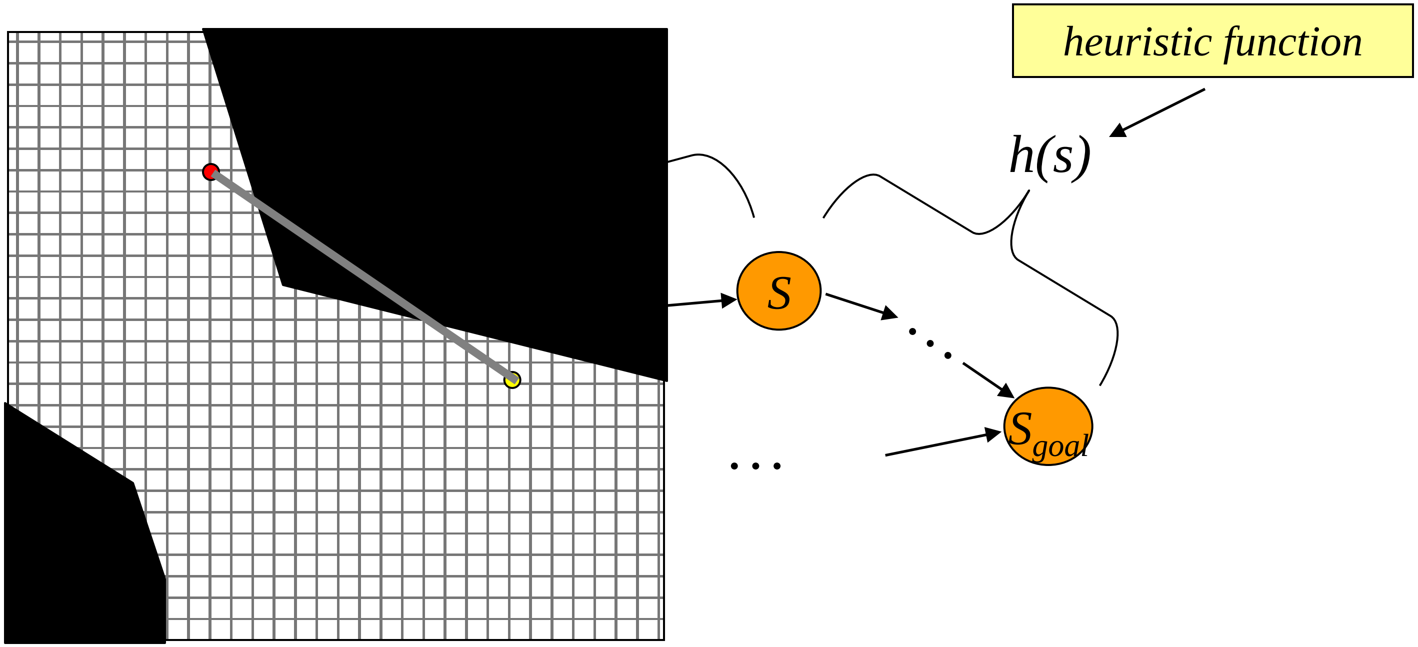
the cost of a shortest path from s_{start} to s ***found so far***

an (under) estimate of the cost of a shortest path from s to s_{goal}

A* Search

- Computes optimal g-values for relevant states

at any point of time:



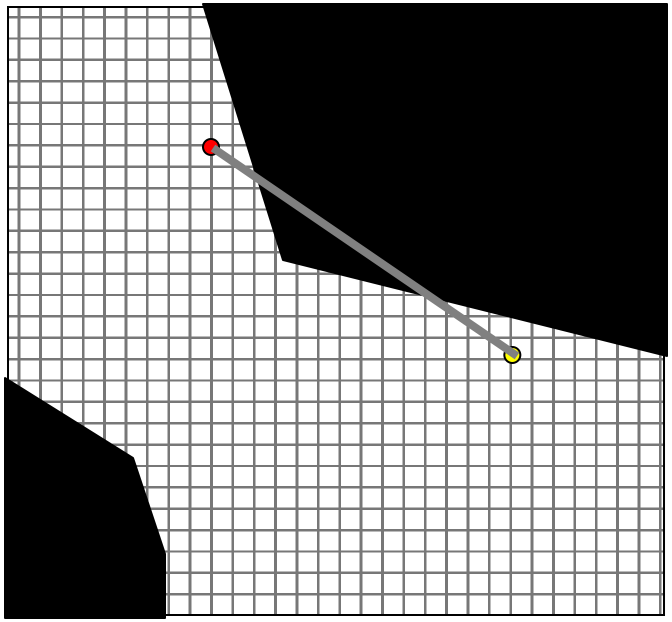
one popular heuristic function – Euclidean distance

A* Search

minimal cost from s to s_{goal}

- Heuristic function must be:
  - admissible: for every state s, $h(s) \leq c^*(s,s_{goal})$
  - consistent (satisfy triangle inequality):

    $h(s_{goal},s_{goal}) = 0$ and for every $s \neq s_{goal}$, $h(s) \leq c(s,succ(s)) + h(succ(s))$
  - admissibility follows from consistency and often consistency follows from admissibility



A* Search

- Computes optimal g-values for relevant states

Main function

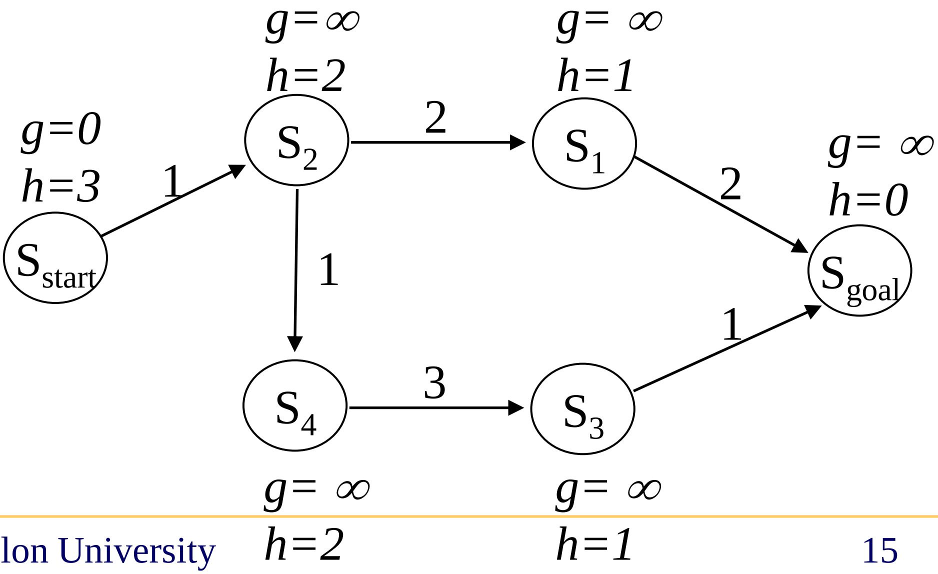
```
$g(s_{start}) = 0$; all other $g$-values are infinite; $OPEN = \{s_{start}\}$;
ComputePath();
publish solution;
```

set of candidates for expansion

ComputePath function

```
while($s_{goal}$ is not expanded)
    remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
    expand $s$;
```

for every expanded state
g(s) is optimal
(if heuristics are consistent)

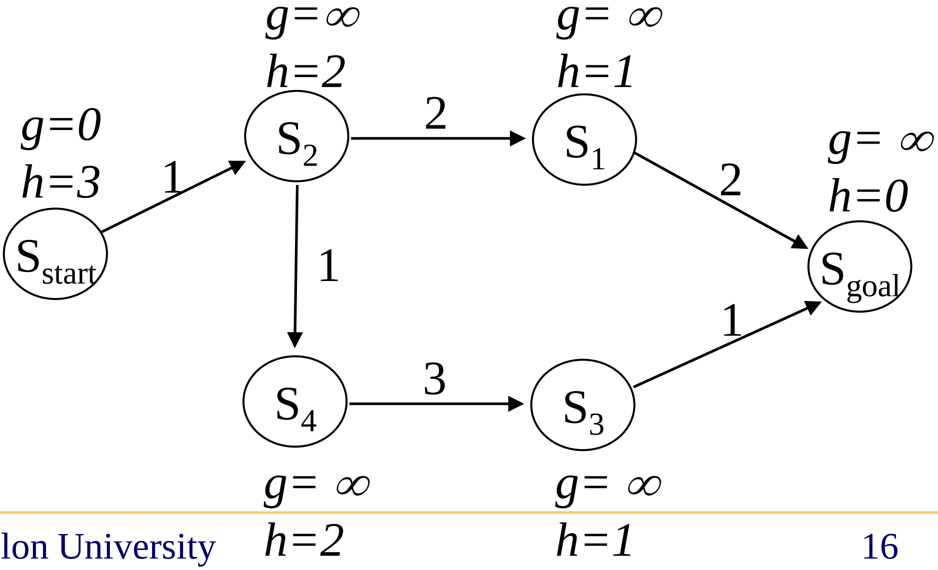


A* Search

- Computes optimal g-values for relevant states

ComputePath function

```
while($s_{goal}$ is not expanded)
  remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
  expand $s$;
```



A* Search

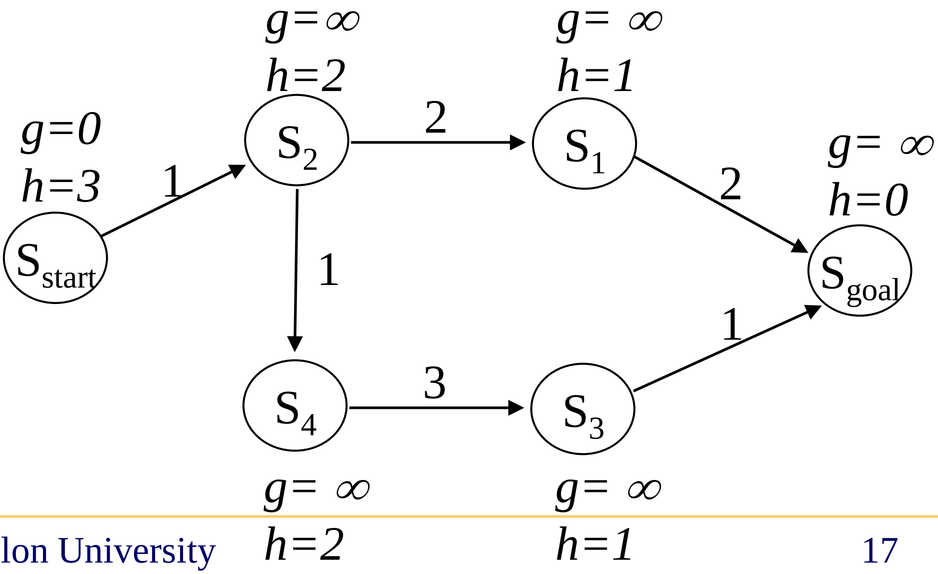
- Computes optimal g-values for relevant states

ComputePath function

```
while($s_{goal}$ is not expanded)
  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 $OPEN$;
```

set of states that have already been expanded

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

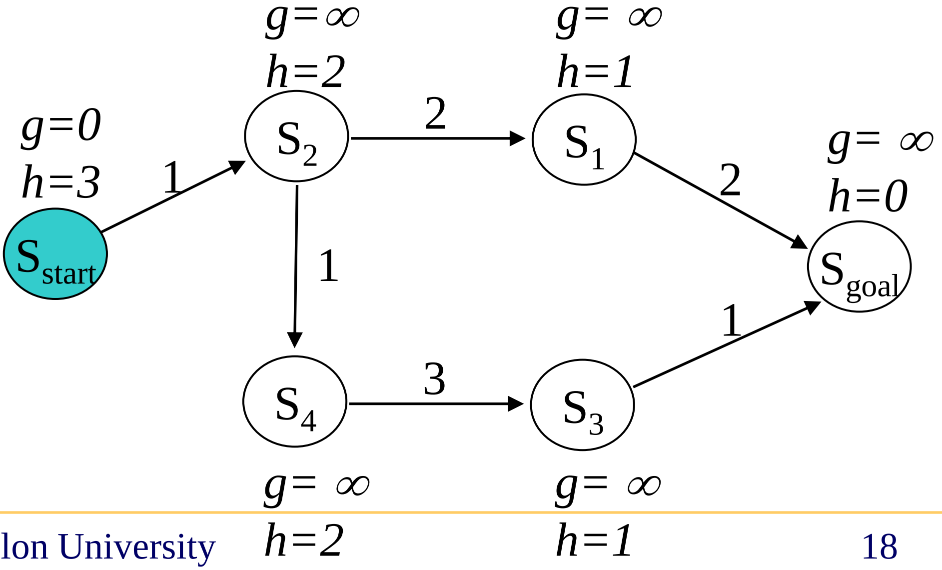


A* Search

- Computes optimal g-values for relevant states

ComputePath function

```
while(s_goal is not expanded)
  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}

A* Search

- Computes optimal g-values for relevant states

ComputePath function

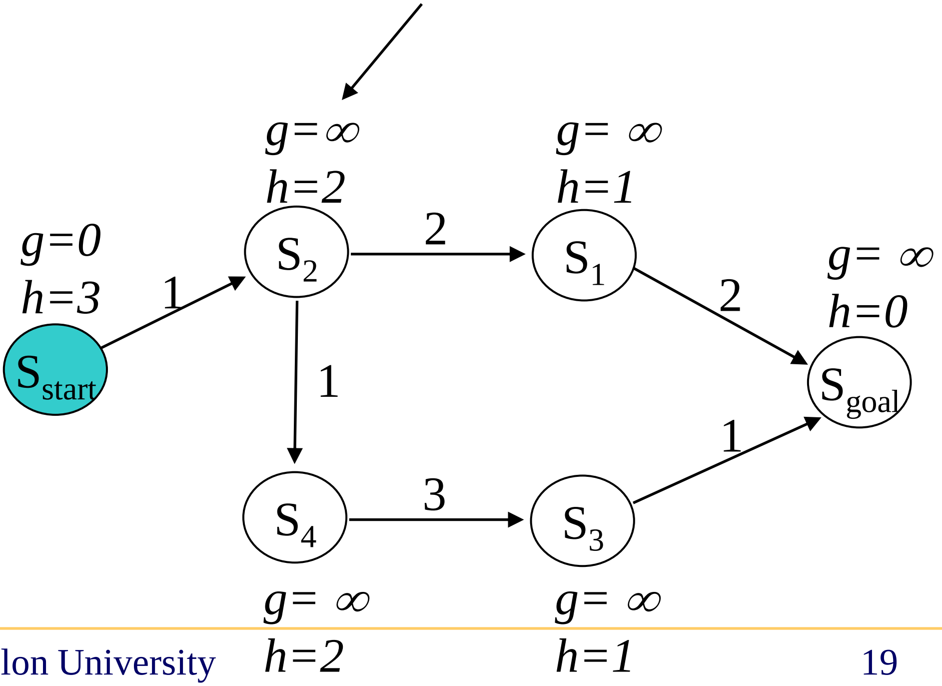
```
while($s_{goal}$ is not expanded)
  remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
  insert $s$ into $CLOSED$;
  for every successor $s'$ of $s$ such that $s'$ not in $CLOSED$
      if $g(s') > g(s) + c(s,s')$
        $g(s') = g(s) + c(s,s')$;
        insert $s'$ into $OPEN$;
```

$CLOSED = \{\}$

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

next state to expand: s_{start}

$g(s_2) > g(s_{start}) + c(s_{start},s_2)$

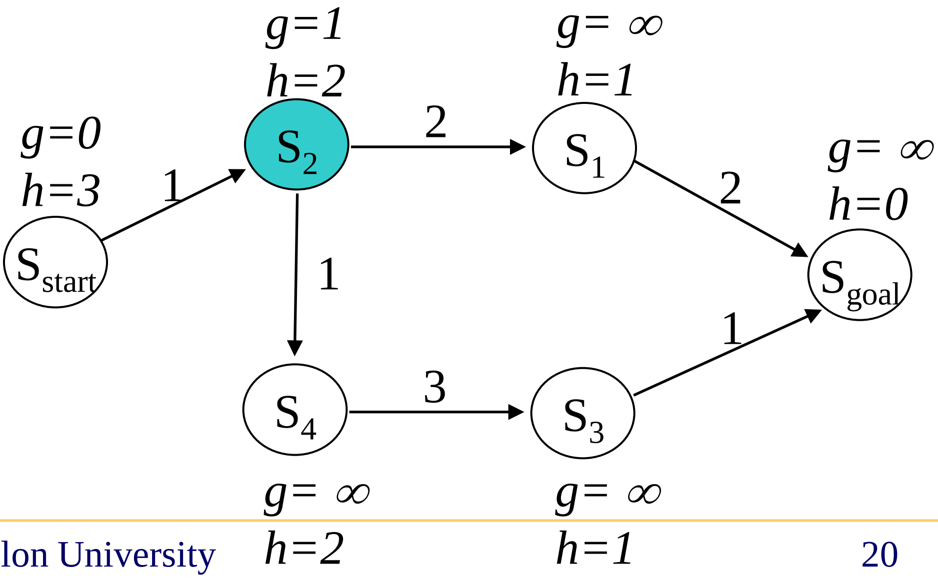


A* Search

- Computes optimal g-values for relevant states

ComputePath function

```
while($s_{goal}$ is not expanded)
  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 $OPEN$;
```



A* Search

- Computes optimal g-values for relevant states

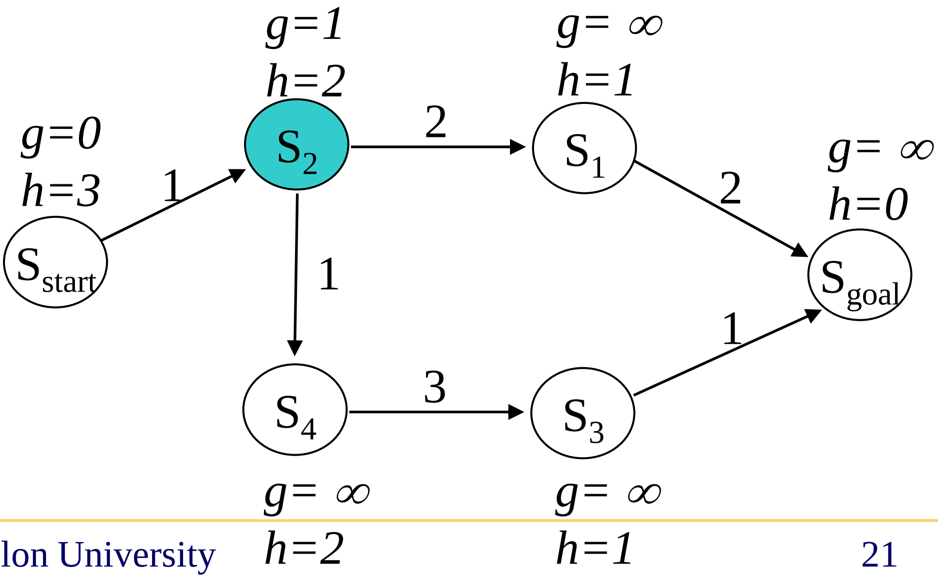
ComputePath function

```
while($s_{goal}$ is not expanded)
    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



A* Search

- Computes optimal g-values for relevant states

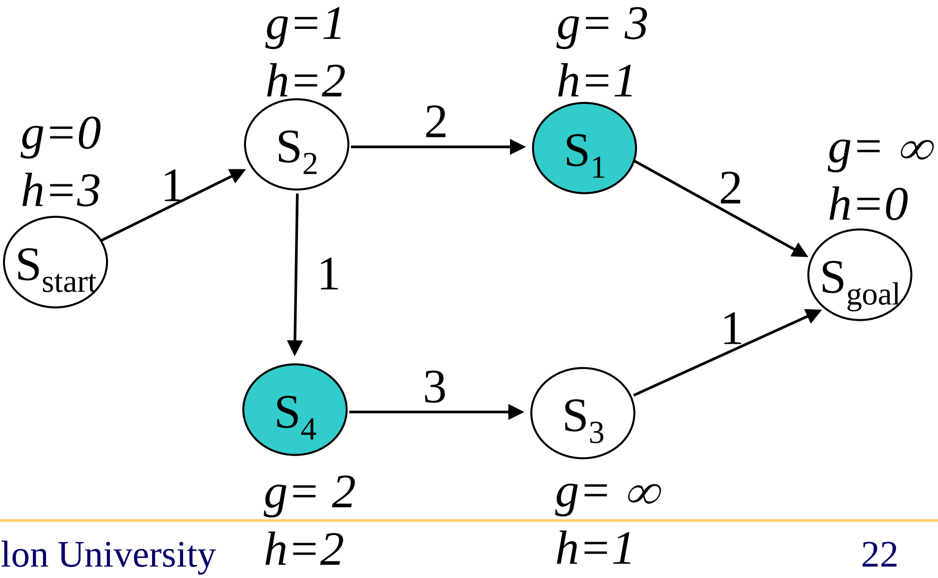
ComputePath function

```
while($s_{goal}$ is not expanded)
  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



A* Search

- Computes optimal g-values for relevant states

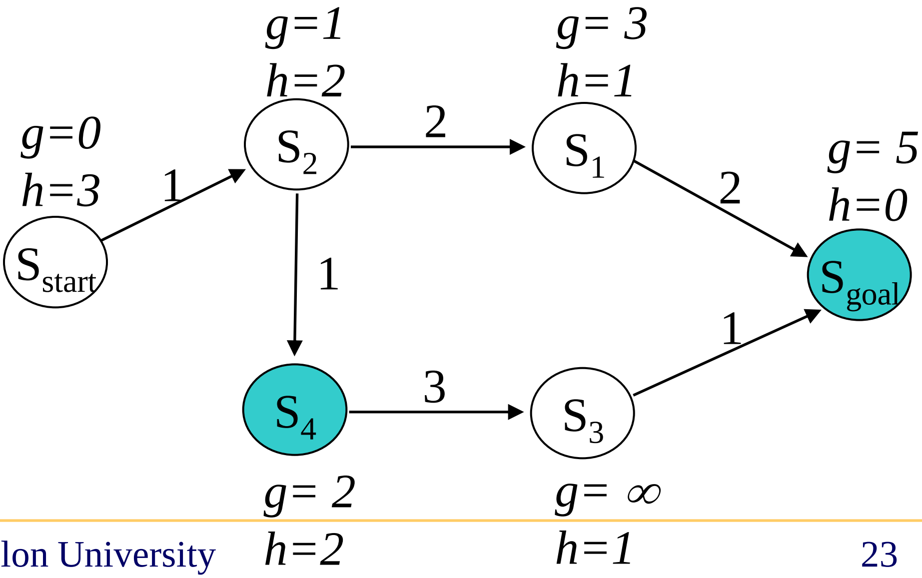
ComputePath function

```
while($s_{goal}$ is not expanded)
  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



A* Search

- Computes optimal g-values for relevant states

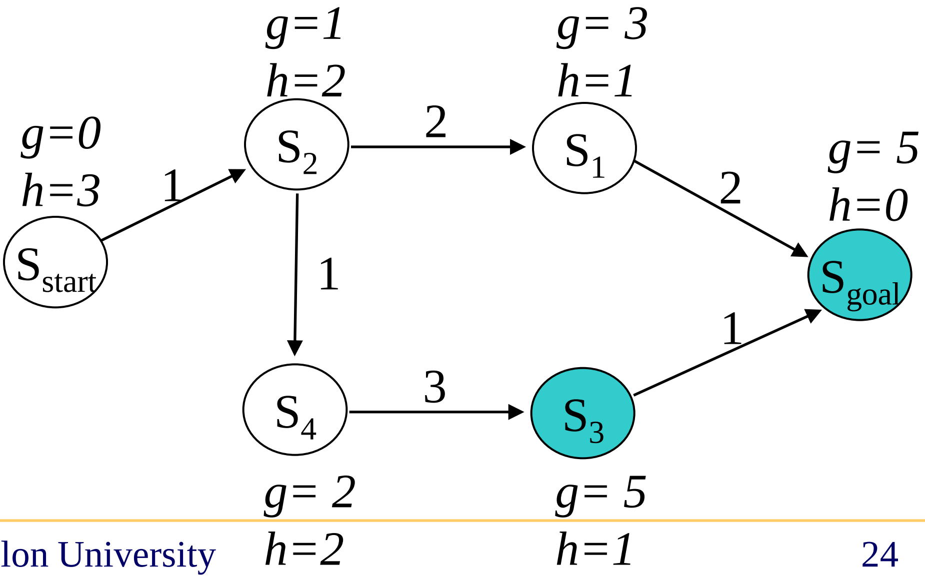
ComputePath function

```
while($s_{goal}$ is not expanded)
   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}



A* Search

- Computes optimal g-values for relevant states

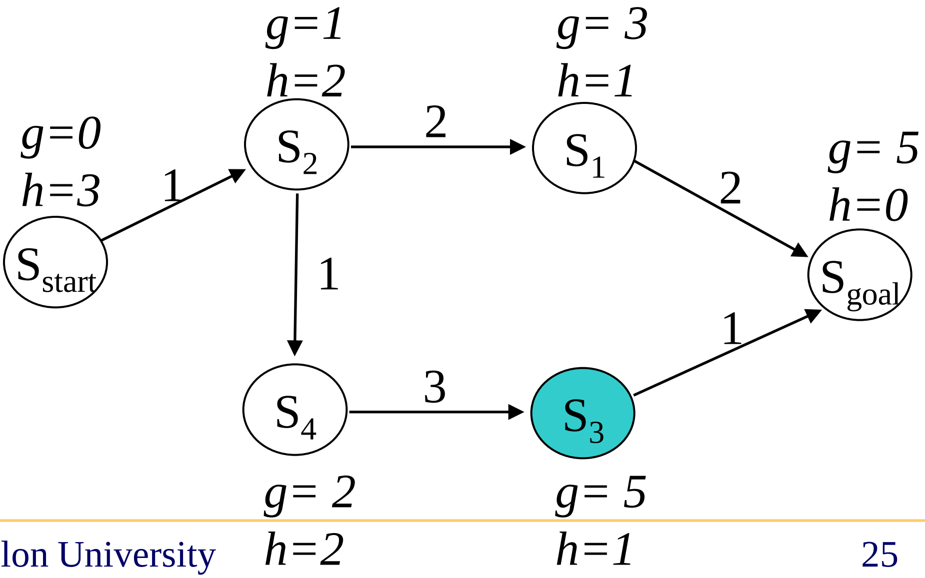
ComputePath function

```
while($s_{goal}$ is not expanded)
   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

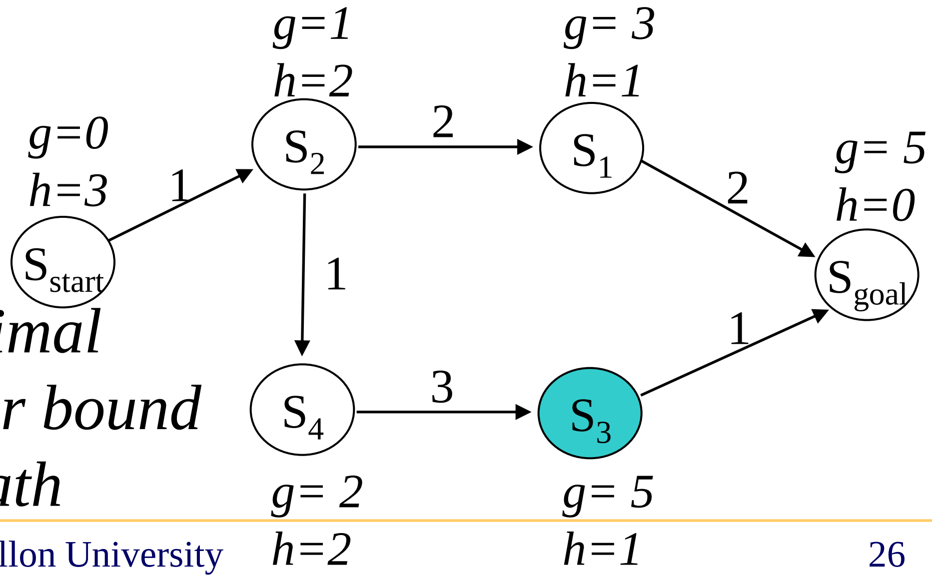


A* Search

- Computes optimal g-values for relevant states

ComputePath function

```
while($s_{goal}$ is not expanded)
    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 $OPEN$;
```



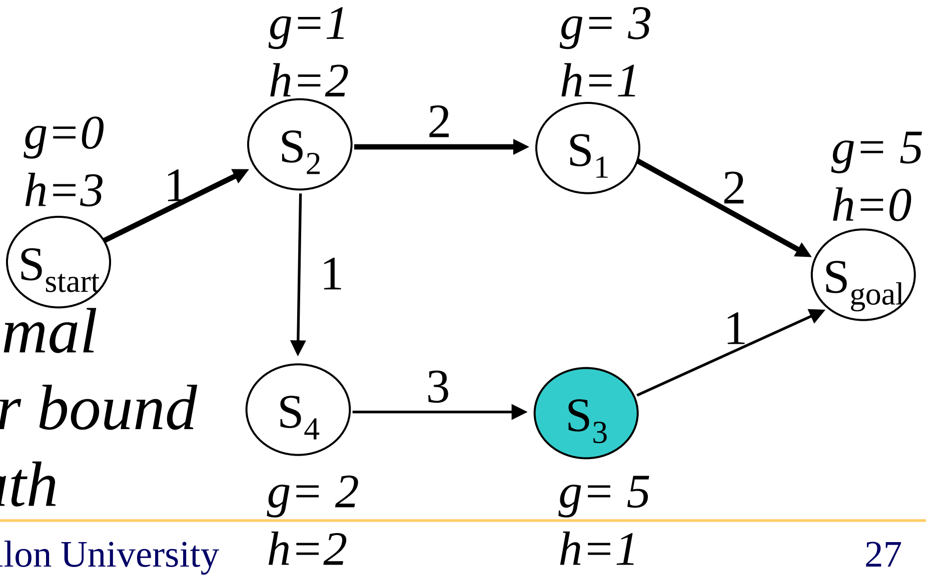
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

A* Search

- Computes optimal g-values for relevant states

ComputePath function

```
while($s_{goal}$ is not expanded)
    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

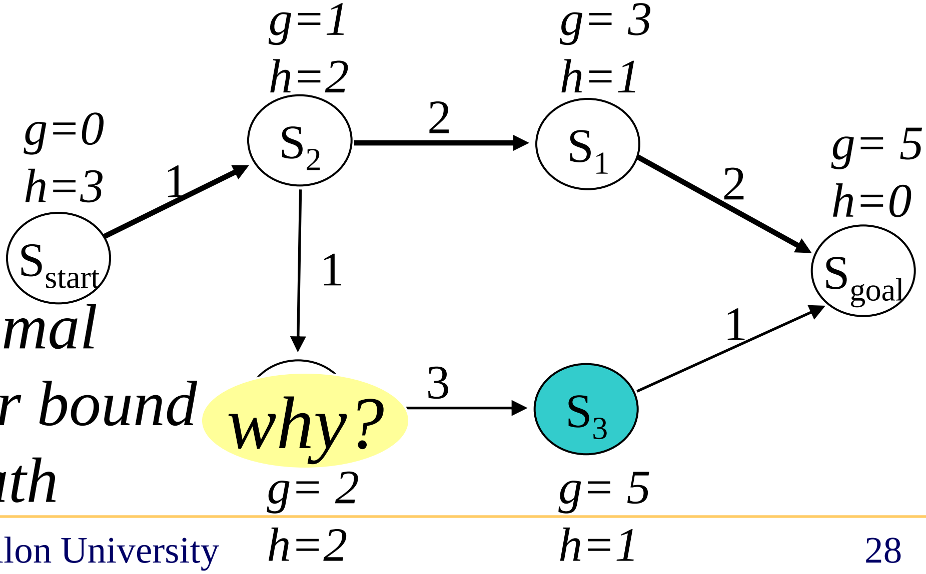
we can now compute a least-cost path

A* Search

- Computes optimal g-values for relevant states

ComputePath function

```
while($s_{goal}$ is not expanded)
    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 $OPEN$;
```



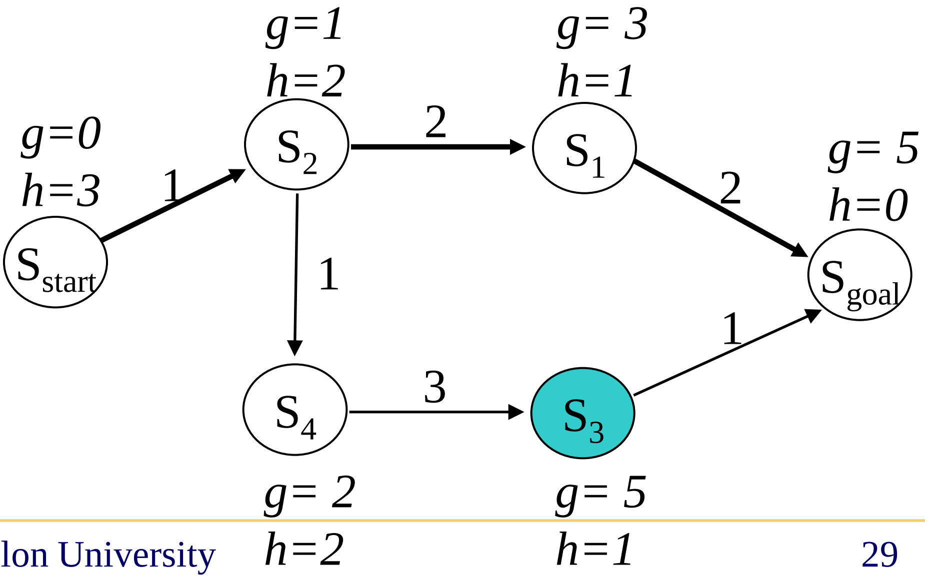
why?

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* Search

- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution

- Performs provably minimal number of state expansions required to guarantee optimality – optimal in terms of the computations



A* Search

- Is guaranteed to return an optimal path (in fact, for every expanded state) – optimal in terms of the solution

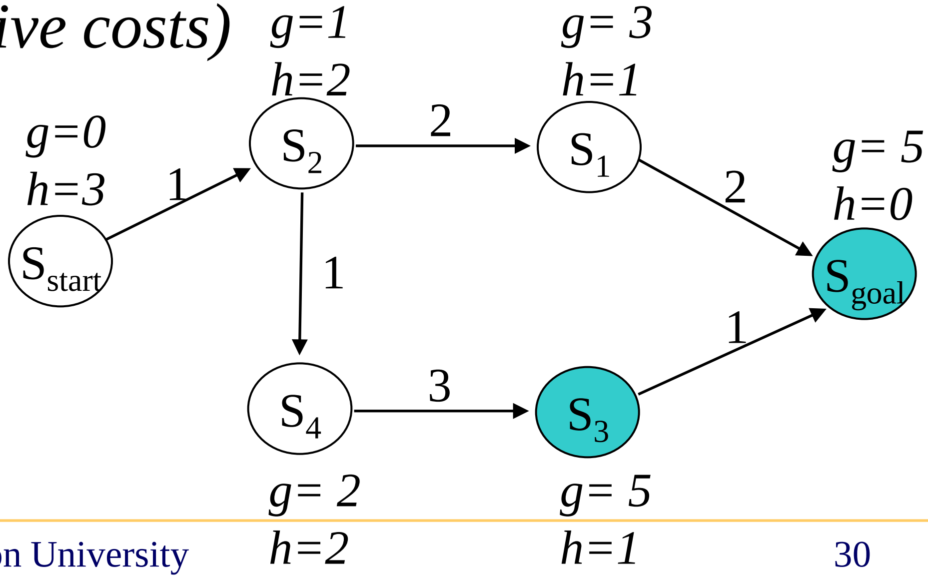
Sketch of proof by induction for h = 0:

assume all previously expanded states have optimal g-values

next state to expand is s: f(s) = g(s) – min among states in OPEN

OPEN separates expanded states from never seen states

thus, path to s via a state in OPEN or an unseen state will be worse than g(s) (assuming positive costs)



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

Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values

ComputePath function

```
while($s_{goal}$ is not expanded)
  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 $OPEN$;
```

expansion of s

Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values

Sketch of proof of optimality by induction for consistent h:

1. *assume all previously expanded states have optimal g-values*
2. *next state to expand is s: $f(s) = g(s)+h(s)$ – min among states in OPEN*
3. *assume $g(s)$ is suboptimal*
4. *then there must be at least one state s' on an optimal path from start to s such that it is in OPEN but wasn't expanded*
5. $g(s') + h(s') \geq g(s)+h(s)$
6. *but* $g(s') + c^*(s',s) < g(s) \Rightarrow$

   $$g(s') + c^*(s',s) + h(s) < g(s) + h(s) \Rightarrow$$

   $$g(s') + h(s') < g(s) + h(s)$$
7. *thus it must be the case that $g(s)$ is optimal*

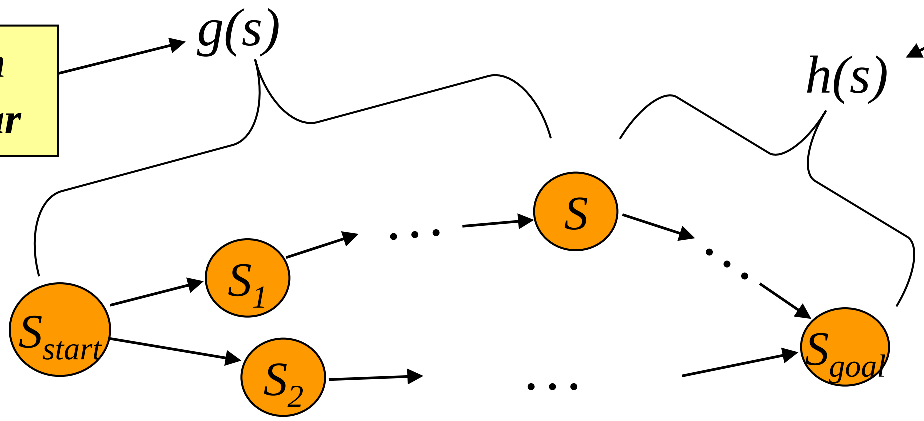
Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values
- Dijkstra's: expands states in the order of $f = g$ values (pretty much)

- Intuitively: f(s) – estimate of the cost of a least cost path from start to goal via s

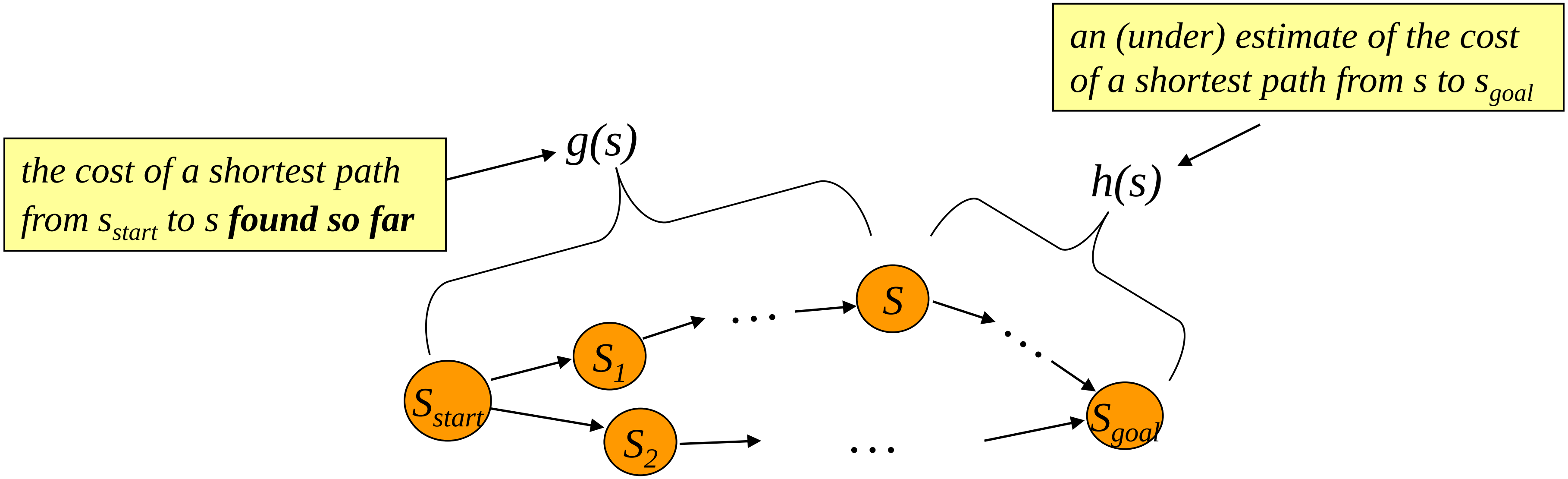
*the cost of a shortest path from s_{start} to s **found so far***

an (under) estimate of the cost of a shortest path from s to s_{goal}



Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values
- Dijkstra's: expands states in the order of $f = g$ values (pretty much)
- **Weighted A*:** expands states in the order of $f = g+\varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal



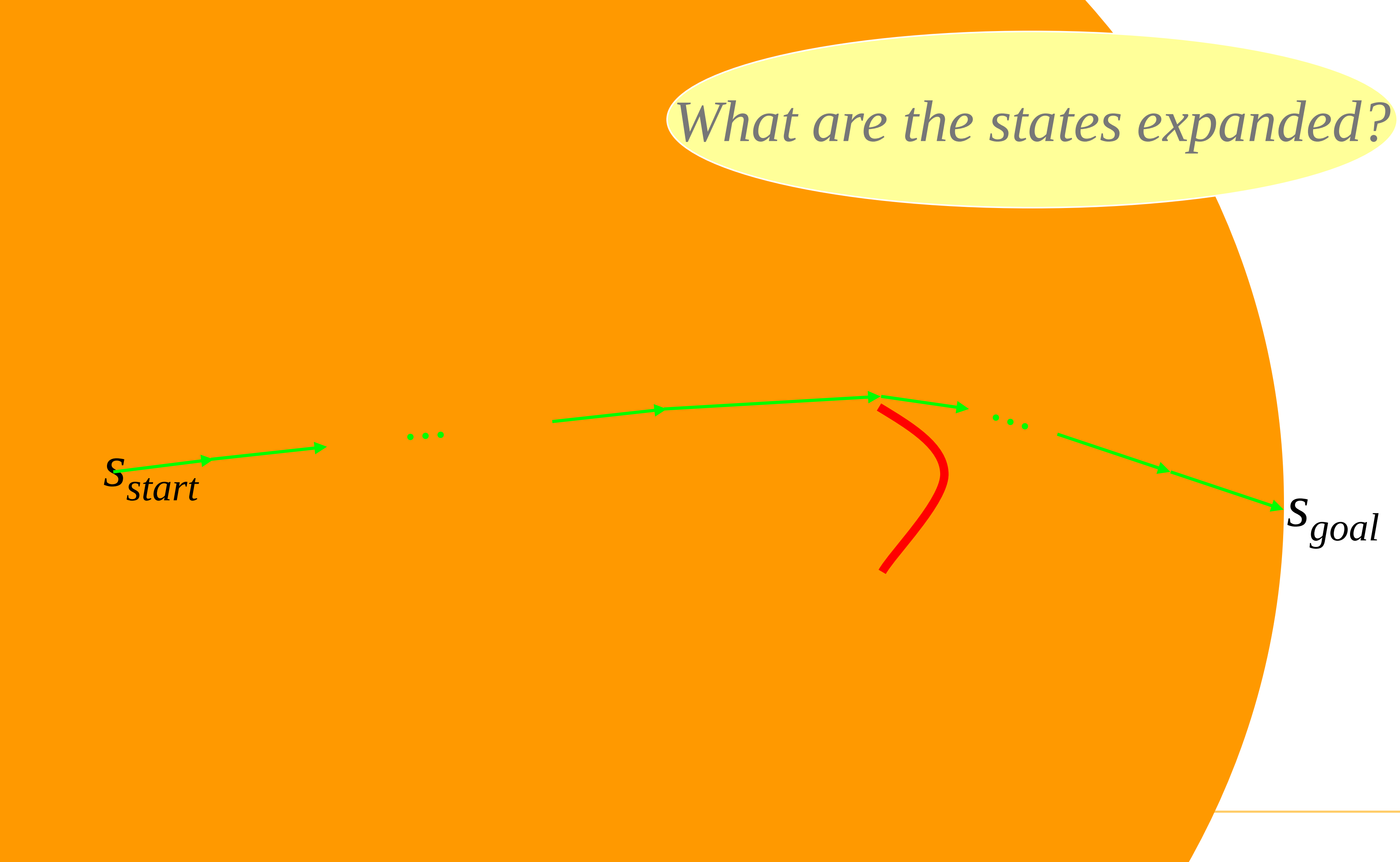
an (under) estimate of the cost of a shortest path from s to s_{goal}

the cost of a shortest path from s_{start} *to s* ***found so far***

Effect of the Heuristic Function

- Dijkstra’s: expands states in the order of $f = g$ values

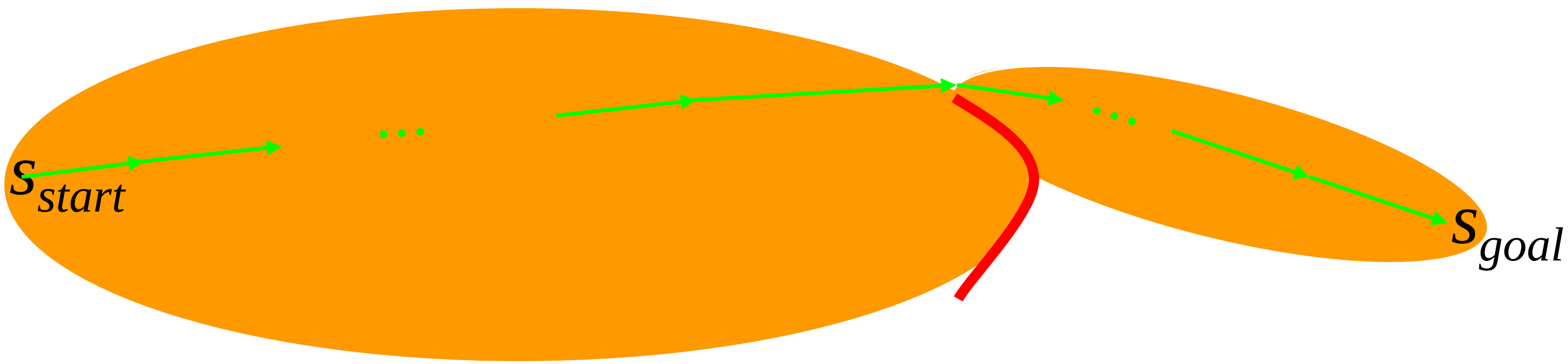
What are the states expanded?



Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values

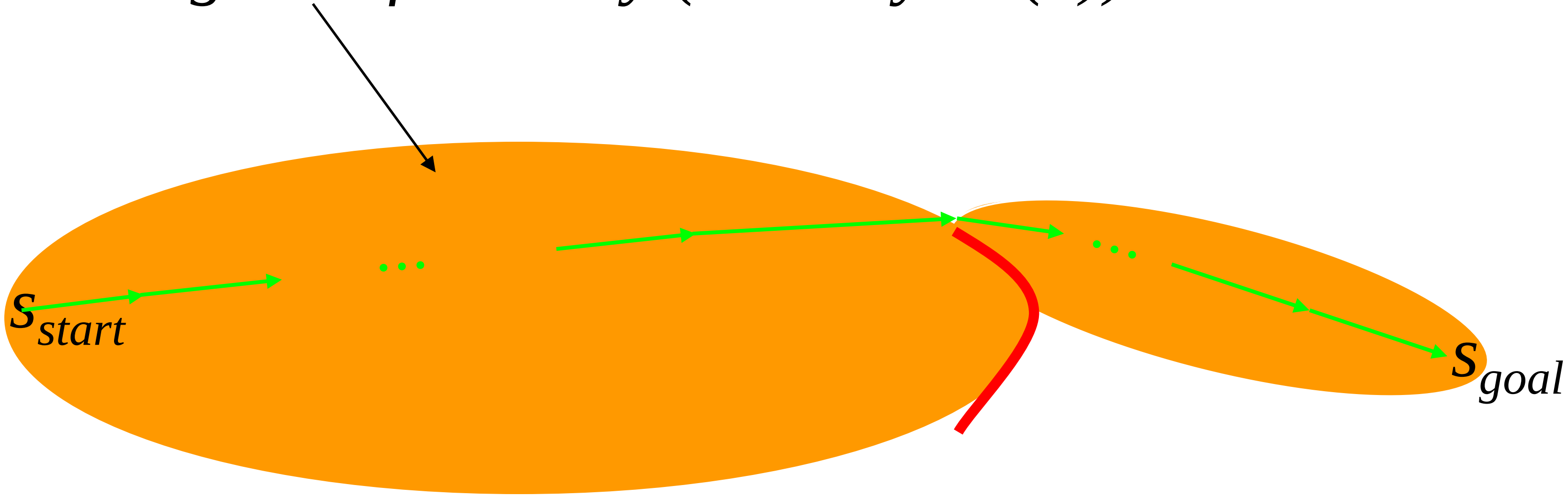
What are the states expanded?



Effect of the Heuristic Function

- A* Search: expands states in the order of $f = g+h$ values

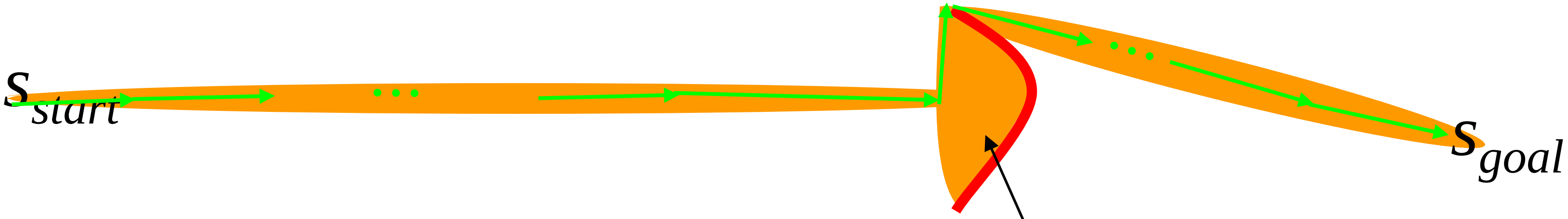
for large problems this results in A quickly running out of memory (memory: O(n))*



Effect of the Heuristic Function

- Weighted A* Search: expands states in the order of $f = g+\varepsilon h$ values, $\varepsilon > 1$ = bias towards states that are closer to goal

what states are expanded?
– research question



key to finding solution fast:
shallow minima for h(s)-h(s) function*

Effect of the Heuristic Function

- Weighted A* Search:
  - trades off optimality for speed
  - ε-suboptimal:

    $cost(solution) \leq \varepsilon \cdot cost(optimal\ solution)$
  - in many domains, it has been shown to be orders of magnitude faster than A*
  - research becomes to develop a heuristic function that has shallow local minima

Effect of the Heuristic Function

- Weighted A* Search:
  - trades off optimality for speed
  - ε-suboptimal:

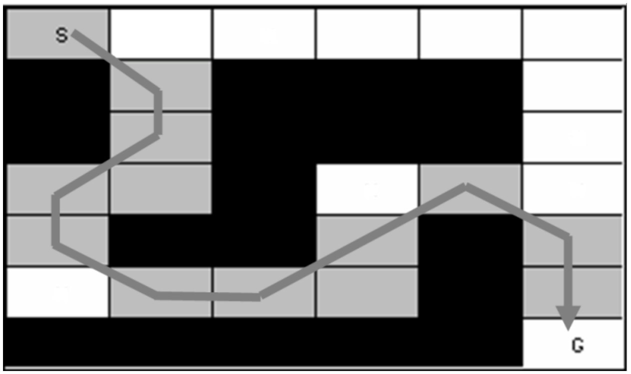
    $cost(solution) \leq \varepsilon \cdot cost(optimal\ so$
  - in many domains, it has bee
    faster than A*
  - research becomes to develop a heuristic function that has shallow local minima

Is it guaranteed to expand no more states than A?*

Effect of the Heuristic Function

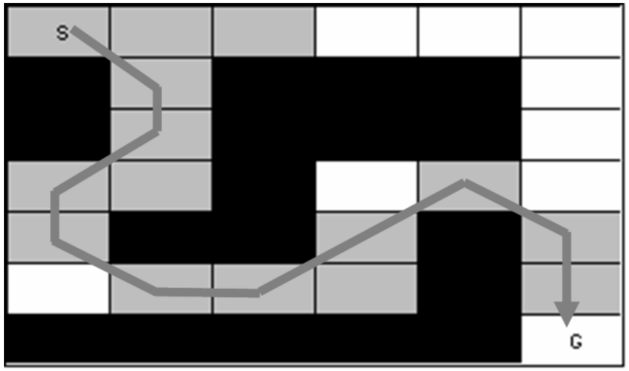
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing *ε:*

ε =2.5



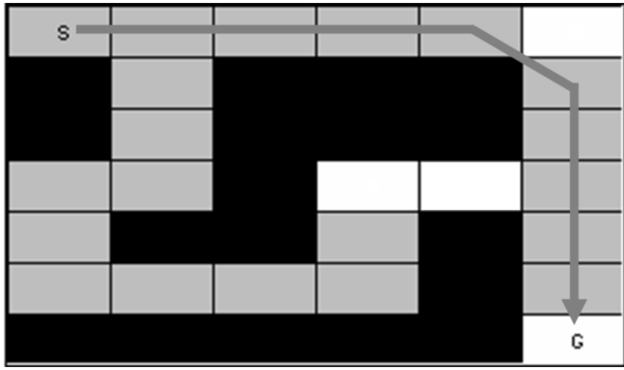
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0

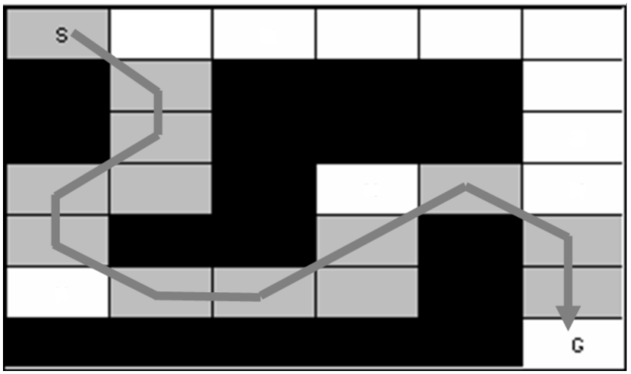


20 expansions
solution=10 moves

Effect of the Heuristic Function

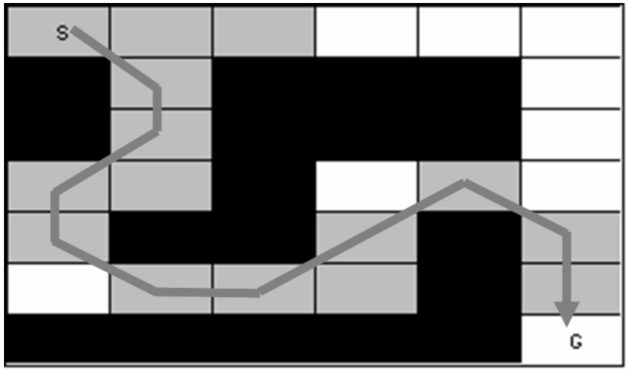
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing ε:

$\varepsilon = 2.5$



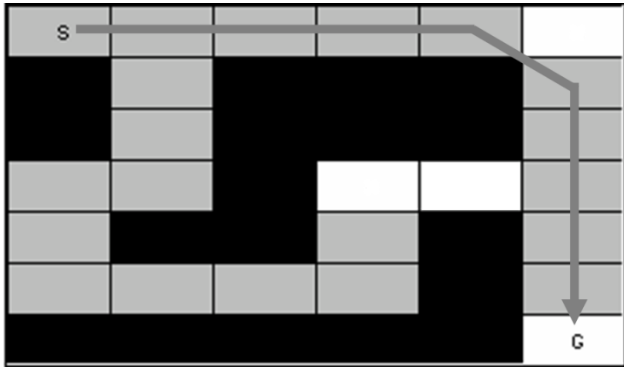
13 expansions
solution=11 moves

$\varepsilon = 1.5$



15 expansions
solution=11 moves

$\varepsilon = 1.0$



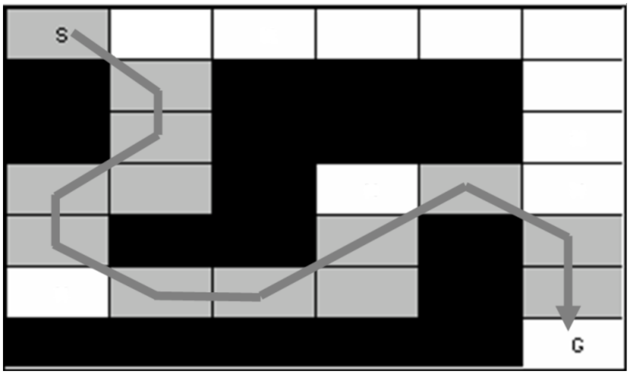
20 expansions
solution=10 moves

- Inefficient because
  - –many state values remain the same between search iterations
  - –we should be able to reuse the results of previous searches

Effect of the Heuristic Function

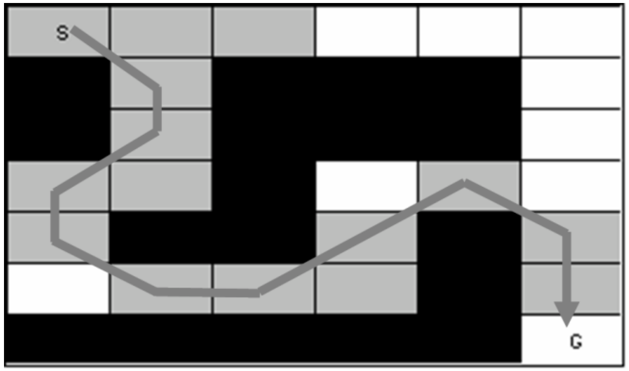
- Constructing anytime search based on weighted A*:
  - find the best path possible given some amount of time for planning
  - do it by running a series of weighted A* searches with decreasing ε:

ε =2.5



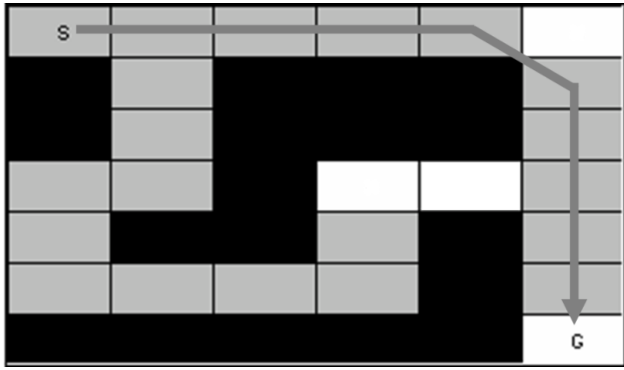
13 expansions
solution=11 moves

ε =1.5



15 expansions
solution=11 moves

ε =1.0



20 expansions
solution=10 moves

- ARA*(will be explained in a later lecture)
  - an efficient version of the above that reuses state values within any search iteration
  - will learn next lecture after we learn about incremental version of A*

Effect of the Heuristic Function

- Useful properties to know:

  - $h_1(s), h_2(s)$ – consistent, then:

    $h(s) = max(h_1(s),h_2(s))$ – consistent

  - if A* uses ε-consistent heuristics:

    $h(s_{goal}) = 0$ *and* $h(s) \leq \varepsilon\, c(s,succ(s)) + h(succ(s)$ *for all* $s \neq s_{goal}$,

    then A* is ε-suboptimal:

    $cost(solution) \leq \varepsilon\, cost(optimal\ solution)$

  - weighted A* is A* with ε-consistent heuristics

  - $h_1(s), h_2(s)$ – consistent, then:

    $h(s) = h_1(s)+h_2(s)$ – ε-consistent

Effect of the Heuristic Function

- Useful properties to know:

  \- $h_1(s)$, $h_2(s)$ – consistent, then:

  $h(s) = max(h_1(s),h_2(s))$ – consistent

  \- if A* uses ε-consistent heuristics:

  $h(s_{goal}) = 0$ *and* $h(s) \leq \varepsilon\ c(s,succ(s)) + h(succ(s)$ *for all* $s \neq s_{goal}$,

  then A* is ε-suboptimal:

  $cost(solution) \leq \varepsilon\ cost(optimal\ solution)$

  \- weighted A* is A* with ε-consistent heuristics

  Proof?

  \- $h_1(s)$, $h_2(s)$ – consistent, then:

  $h(s) = h_1(s)+h_2(s)$ – ε-consistent

Effect of the Heuristic Function

- Useful properties to know:

  - $h_1(s), h_2(s)$ – consistent, then:

    $h(s) = max(h_1(s),h_2(s))$ – consistent

  - if A* uses ε-consistent heuristics:

    $h(s_{goal}) = 0$ *and* $h(s) \leq \varepsilon\ c(s,succ(s)) + h(succ(s)$ *for all* $s \neq s_{goal}$,

    then A* is ε-suboptimal:

    $cost(solution) \leq \varepsilon\ cost(optimal\ solution)$

  - weighted A* is A* with ε-consistent heuristics

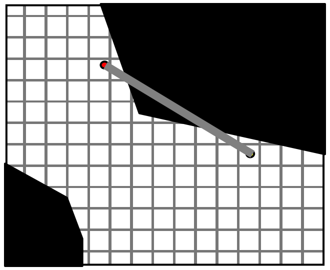
Proof?

  - $h_1(s), h_2(s)$ – consistent, then:

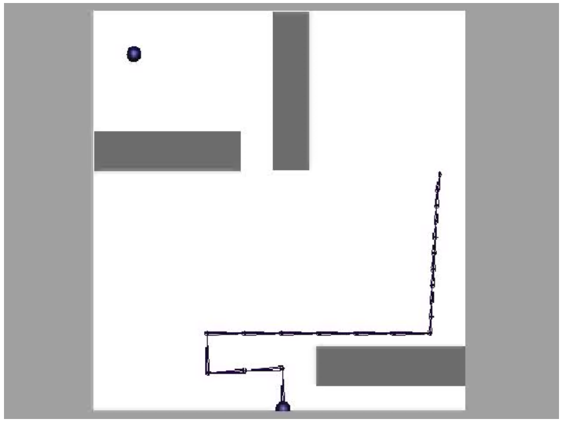
    $h(s) = h_1(s)+h_2(s)$ – ε-consistent

What is ε? Proof?

Examples of Heuristic Function

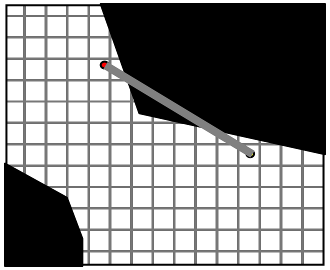


- For grid-based navigation:
  - Euclidean distance
  - Manhattan distance: $h(x,y) = abs(x-x_{goal}) + abs(y-y_{goal})$
  - Diagonal distance: $h(x,y) = max(abs(x-x_{goal}), abs(y-y_{goal}))$
  - More informed distances???



- Robot arm planning:
  - End-effector distance
  - Any others???

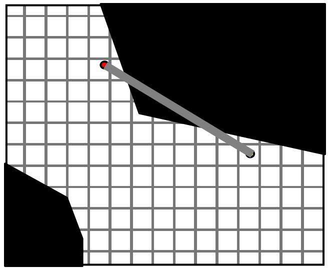
Examples of Heuristic Function



- For grid-based navigation:
  - Euclidean distance
  - Manhattan distance: $h(x,y) = abs(x-x_{goal}) + abs(y-y_{goal})$
  - Diagonal distance: $h(x,y) = max(abs(x-x_{goal}), abs(y-y_{goal}))$
  - More informed distances???

- Robot arm planning:
  - End-effector distance
  - Any others???

Examples of Heuristic Function



- For grid-based navigation:
  - Euclidean distance
  - Manhattan distance: $h(x,y) = abs(x\text{-}x_{goal}) + abs(y\text{-}y_{goal})$
  - Diagonal distance: $h(x,y) = max(abs(x\text{-}x_{goal}), abs(y\text{-}y_{goal}))$
  - More informed distances???

- Autonomous door opening:
  - Heuristic function???

Memory Issues

- A* does provably minimum number of expansions (O(n)) for finding a provably optimal solution

- Memory requirements of A* (O(n)) can be improved though

- Memory requirements of weighted A* are often but not always better

Memory Issues

- Alternatives:
  - Depth-First Search (w/o coloring all expanded states):
    - explore each every possible path at a time avoiding looping and keeping in the memory only the best path discovered so far
    - Complete and optimal (assuming finite state-spaces)
    - Memory: $O(bm)$, where b – max. branching factor, m – max. pathlength
    - Complexity: $O(b^m)$, since it will repeatedly re-expand states

Memory Issues

- Alternatives:
  - Depth-First Search (w/o coloring all expanded states):
    - explore each every possible path at a time avoiding looping and keeping in the memory only the best path discovered so far
    - Complete and optimal (assuming finite state-spaces)
    - Memory: $O(bm)$, where b – max. branching factor, m – max. pathlength
    - Complexity: $O(b^m)$, since it will repeatedly re-expand states
    - Example:
      - graph: a 4-connected grid of 40 by 40 cells, start: center of the grid
      - A* expands up to 800 states, DFS may expand way over $4^{20} > 10^{12}$ states

Memory Issues

- Alternatives:
  - Depth-First Search (w/o coloring all expanded states):
    - explore each every possible path at a time avoiding looping and keeping in the memory only the best path discovered so far
    - Complete and optimal (assuming finite state-spaces)
    - Memory: $O(bm)$, where b – max
    - Complexity: $O(b^m)$, since it wi
    - Example:
      - graph: a 4-connected grid of 40 by 40 cells, start: center of the grid
      - A* expands up to 800 states, DFS may expand way over $4^{20} > 10^{12}$ states

What if goal is few steps away in a huge state-space?

Memory Issues

- Alternatives:
  - IDA* (Iterative Deepening A*)
    1. set $f_{max} = 1$ (or some other small value)
    2. execute (previously explained) DFS that does not expand states with $f > f_{max}$
    3. If DFS returns a path to the goal, return it
    4. Otherwise $f_{max} = f_{max} + 1$ (or larger increment) and go to step 2

Memory Issues

- Alternatives:
  - IDA* (Iterative Deepening A*)
    1. set $f_{max} = 1$ (or some other small value)
    2. execute (previously explained) DFS that does not expand states with $f > f_{max}$
    3. If DFS returns a path to the goal, return it
    4. Otherwise $f_{max} = f_{max} + 1$ (or larger increment) and go to step 2

    - Complete and optimal in any state-space (with positive costs)

    - Memory: $O(bl)$, where b – max. branching factor, l – length of optimal path

    - Complexity: $O(kb^l)$, where k is the number of times DFS is called