Basic Path Finding

Path Planning

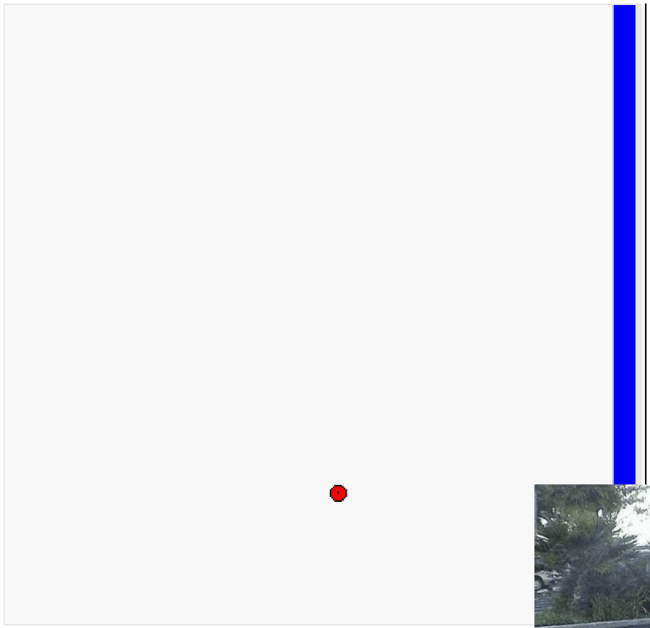
Why not pre-compute all paths?

- Path Planning
  - needs to be very fast (especially for games with many characters)
  - needs to generate believable paths

Path Planning

- Path Planning in
  - partially-known environments is a repeated process
  - dynamic environments is also a repeated process

re-planning path as map becomes known



Kbot lab: Nanolathe Complete
Kbot lab: Starting construction
Storm: Under Attack
Storm: Under Attack
Storm: Under Attack

Definition of Path Planning

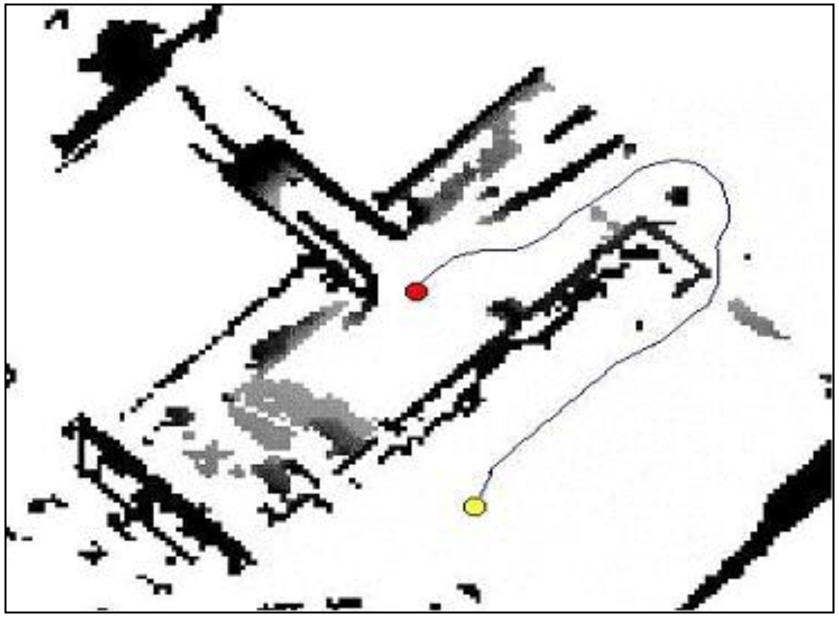
- Task:
  find a feasible (and cost-minimal) path from the current pose to a goal pose

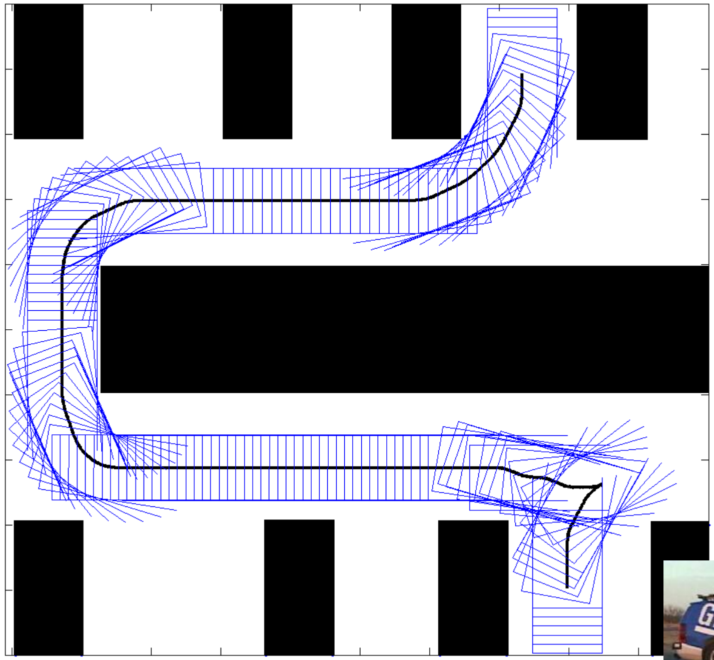
- Two types of constraints:
  environmental constraints (e.g., obstacles)
  dynamics/kinematics constraints

- Generated motion/path should (objective):
  be a feasible path
  minimize cost such as distance, time, unrealistic effects, …

Path Planning

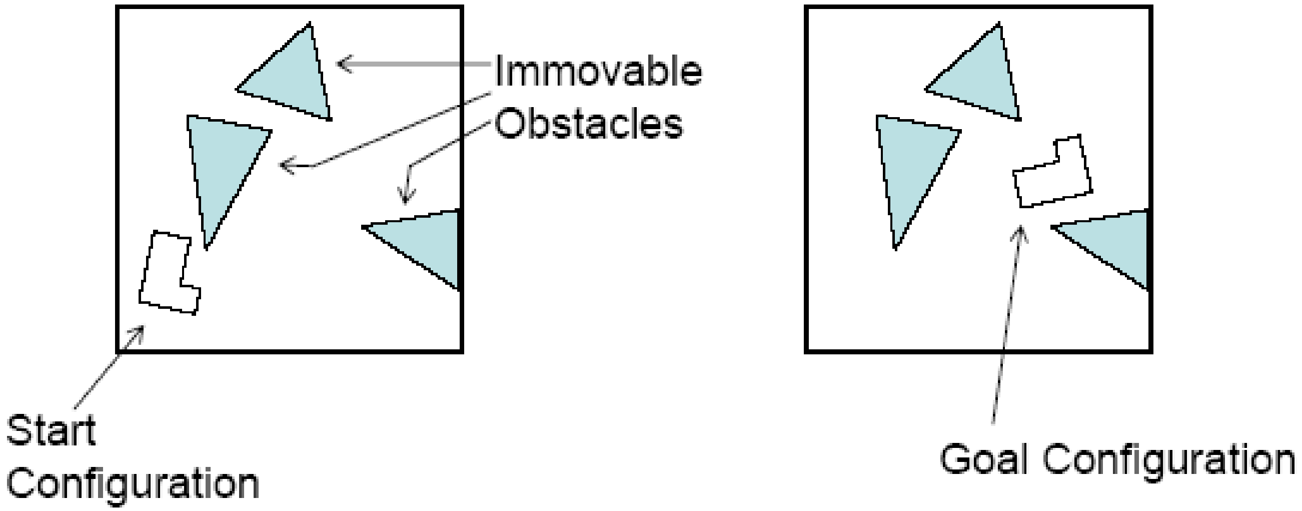
Examples (of what is usually referred to as path planning):





Path Planning

Examples (of what is usually referred to as motion planning):



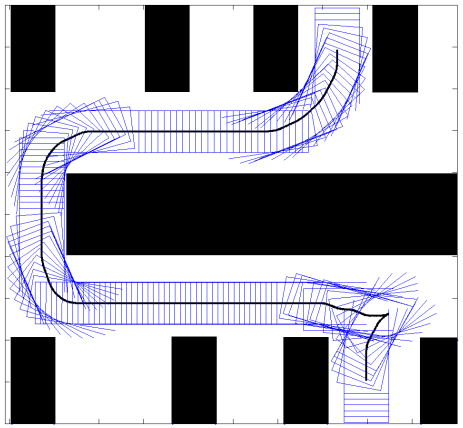
Piano Movers' problem

the example above is borrowed from www.cs.cmu.edu/~awm/tutorials

Configuration Space

- Configuration is legal if it does not intersect any obstacles and is valid (e.g., does not intersect itself, joint angles are within their limits)

- Configuration Space is the set of legal configurations

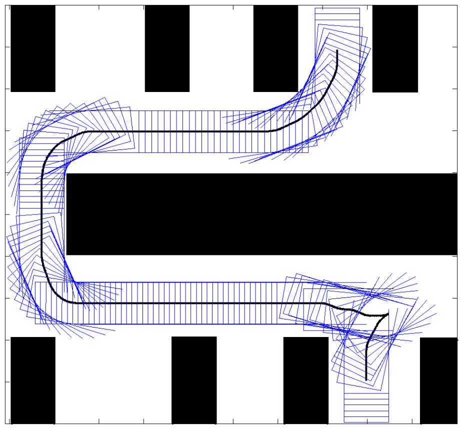


Configuration Space

- Configuration is legal if it does not intersect any obstacles and is valid (e.g., does not intersect itself, joint angles are within their limits)

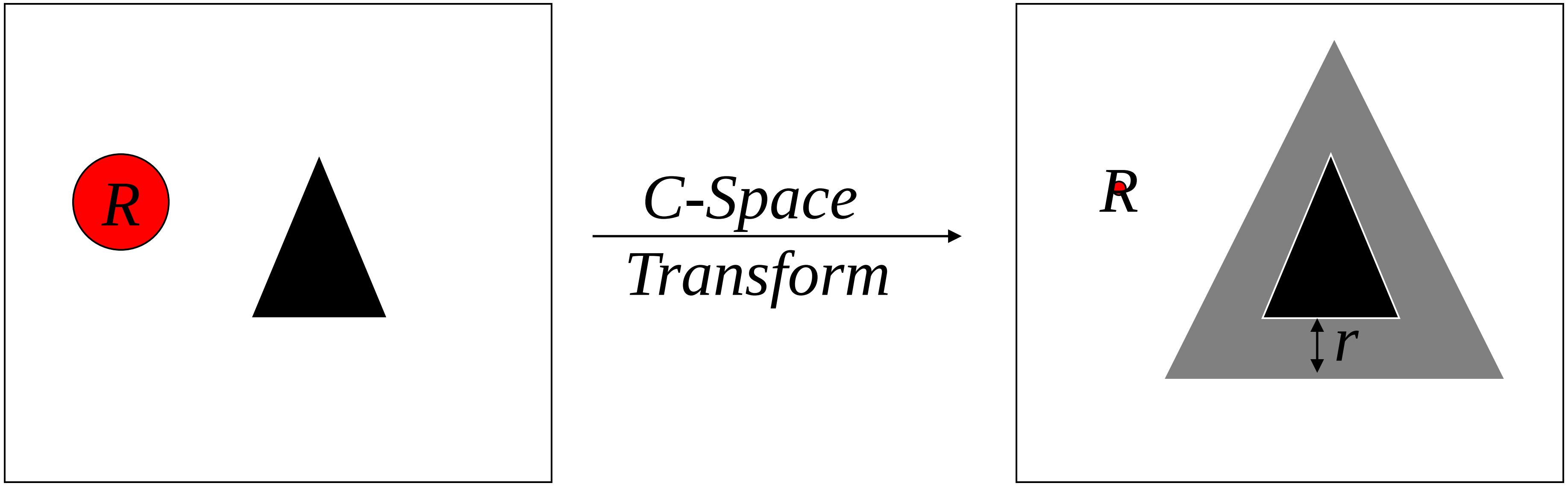
- Configuration Space is the set of legal configurations

What is the dimensionality of this configuration space?



C-Space Transform

- Configuration space for rigid-body objects in 2D world is:
  - 2D if object is circular

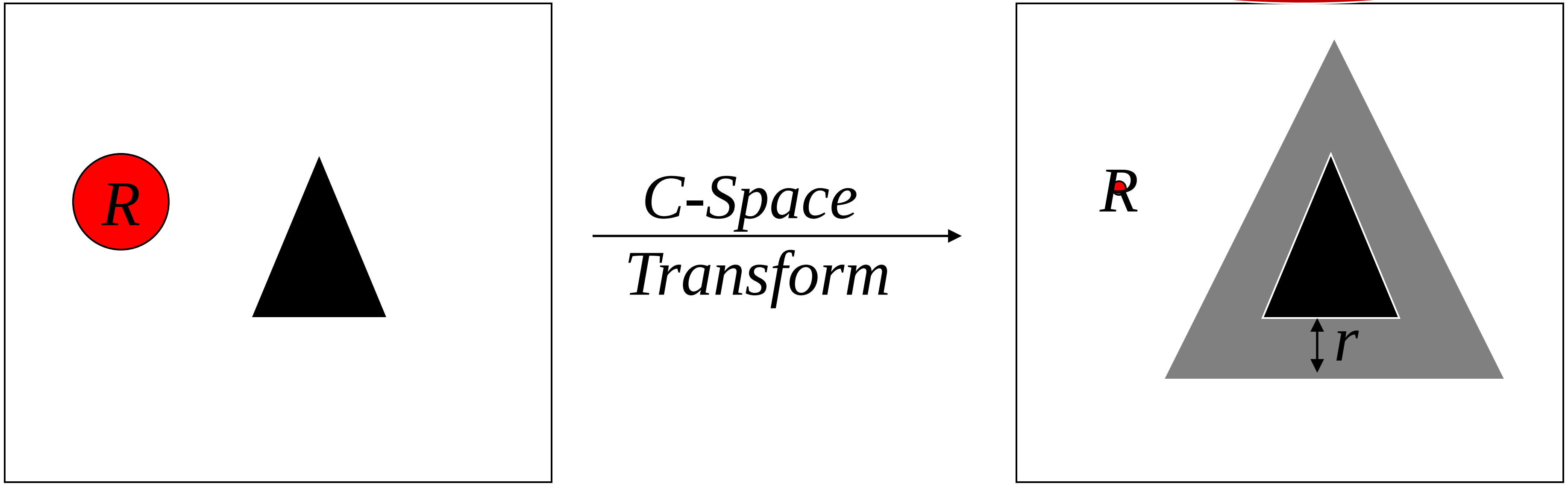


- expand all obstacles by the radius of the object r
- planning can be done for a point R (and not a circle anymore)

C-Space Transform

- Configuration space for rigid-body objects in 2D world is:
  - 2D if object is circular

Is this a correct expansion?

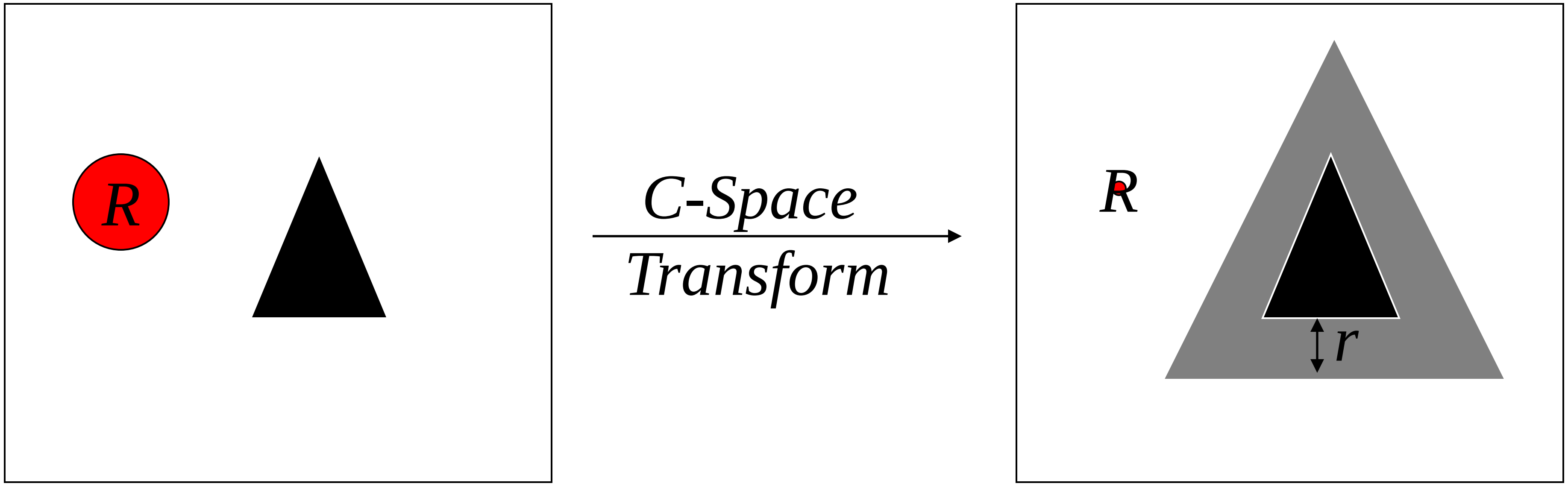


- expand all obstacles by the radius of the object r
- planning can be done for a point R (and not a circle anymore)

C-Space Transform

• Configuration space for rigid-body objects in 2D world is:

- 2D if object is circular

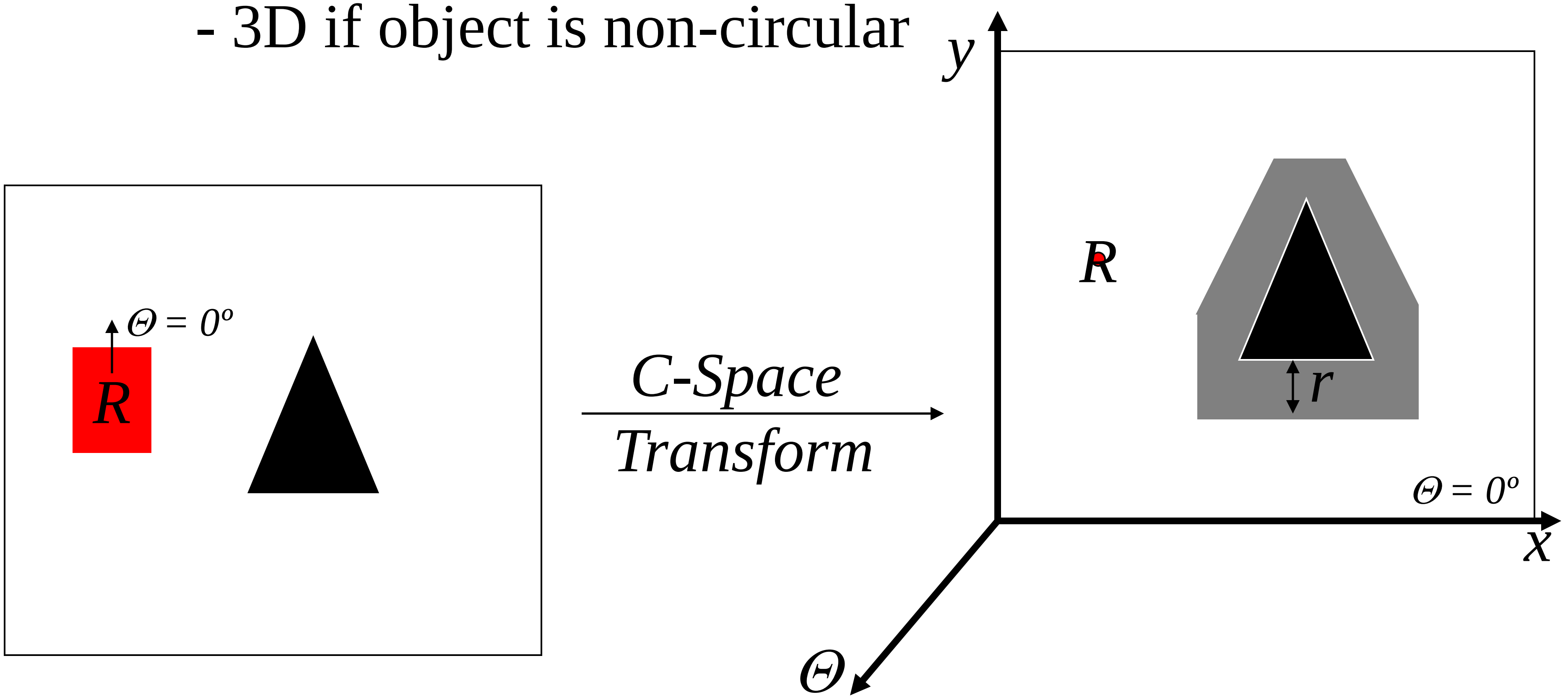


• advantage: planning is faster for a single point *why?*

• disadvantage: need to expand obstacles every time map is updated (O(n) methods exist to compute distance transforms)

C-Space Transform

- Configuration space for arbitrary objects in 2D world is:
  - 3D if object is non-circular



- advantage: planning is faster for a single point
- disadvantage: constructing C-space is expensive

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

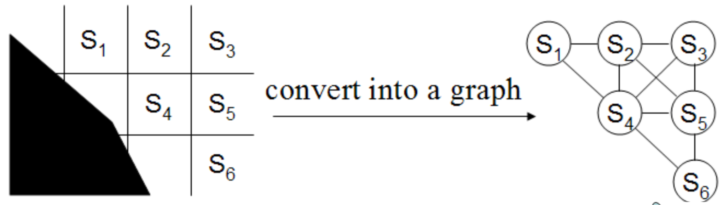
1. Construct a graph representing the planning problem

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

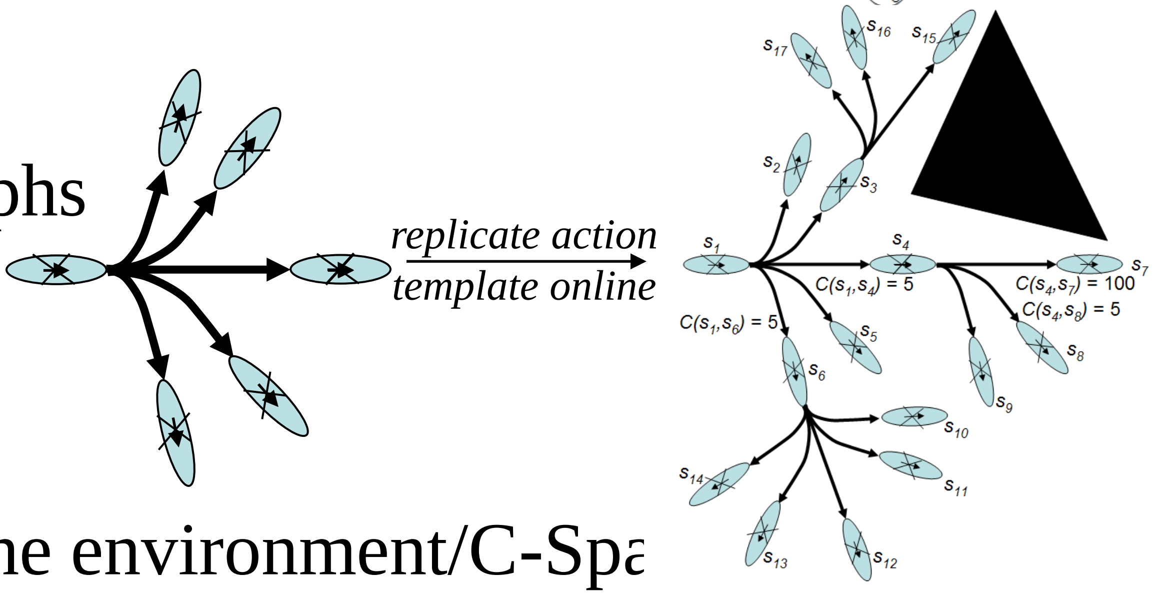
The two steps above are often interleaved

Graph Construction

- Cell decomposition
    - X-connected grids



    - lattice-based graphs

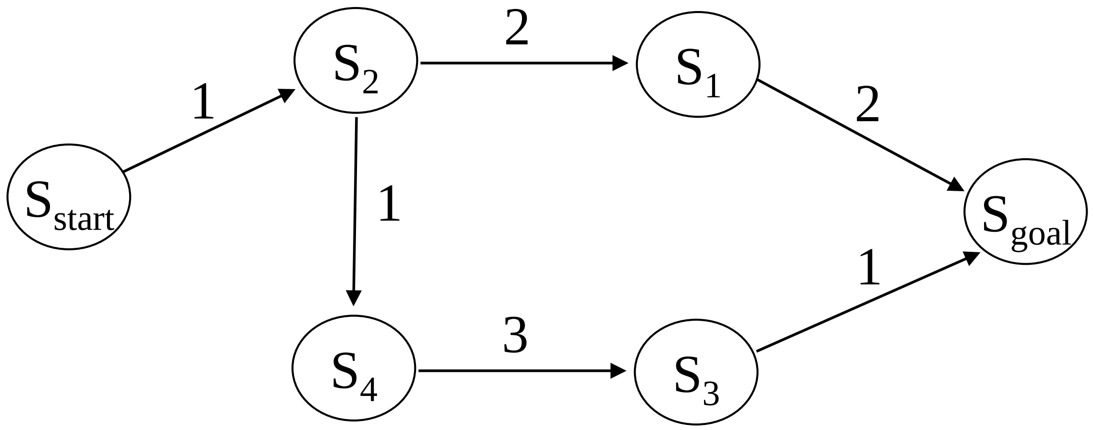


- Skeletonization of the environment/C-Spa
    - -Visibility graphs
    - Voronoi diagrams
    - Probabilistic roadmaps
    - Navmeshes

Graphs Construction

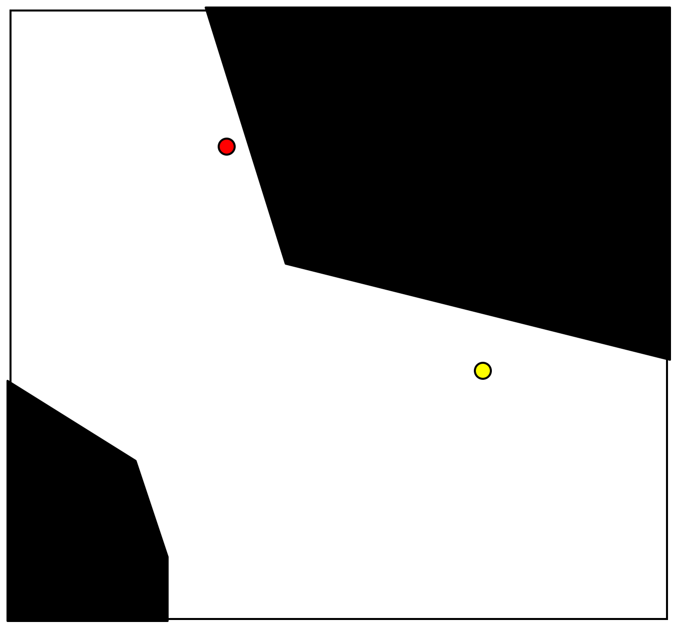
- Once a graph is constructed, we will search it for a least-cost path

- Once again: depending on the planning algorithm, graph construction can be interleaved with graph search



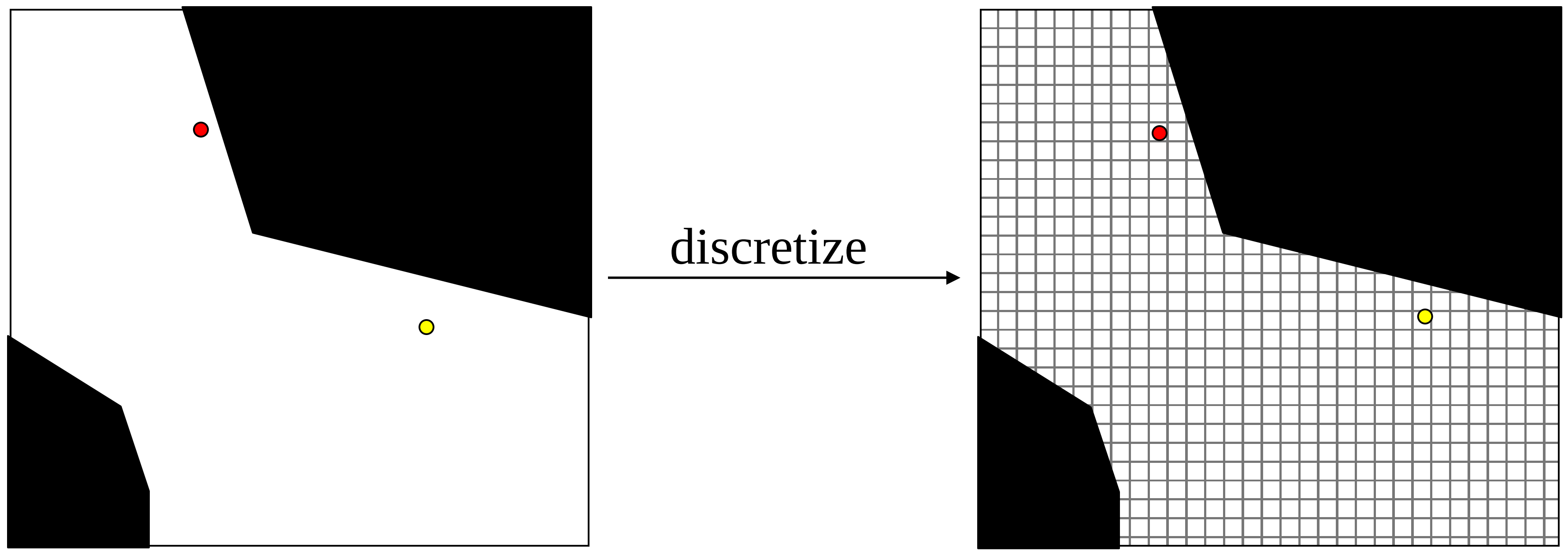
Planning via Cell Decomposition

- Exact Cell Decomposition:
  - overlay convex exact polygons over the free C-space
  - construct a graph, search the graph for a path
  - overly expensive for non-trivial environments and/or above 2D



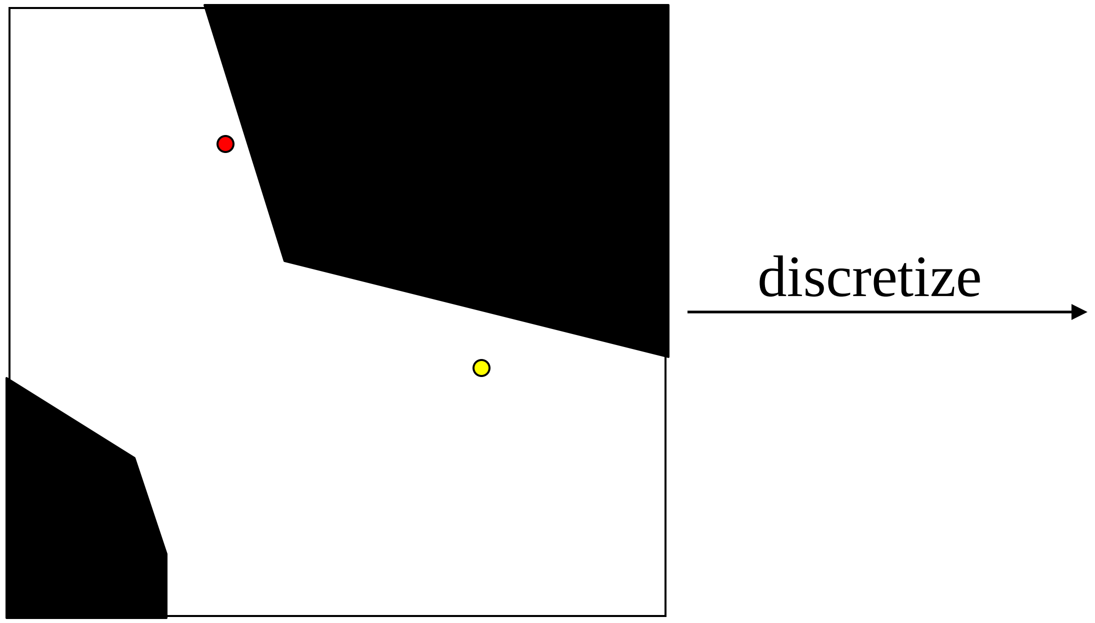
Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - overlay uniform grid over the C-space (discretize)

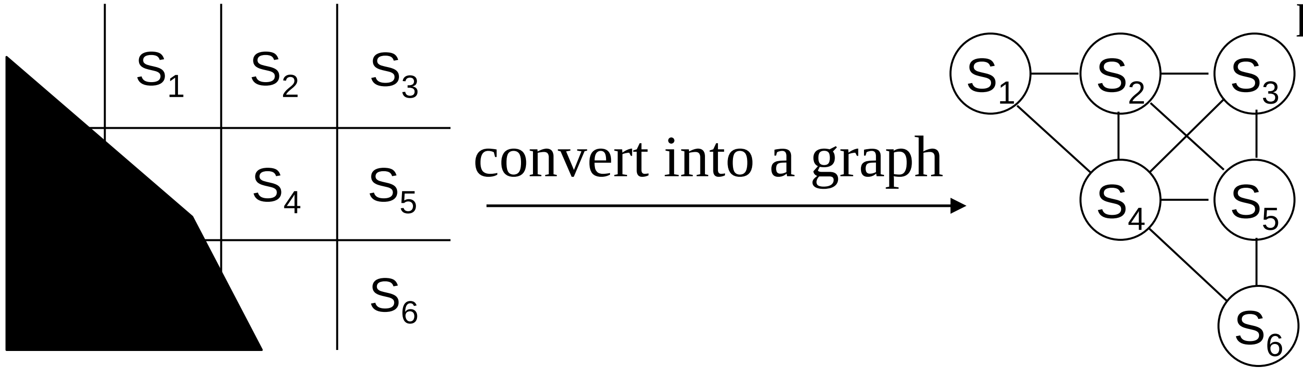


Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path



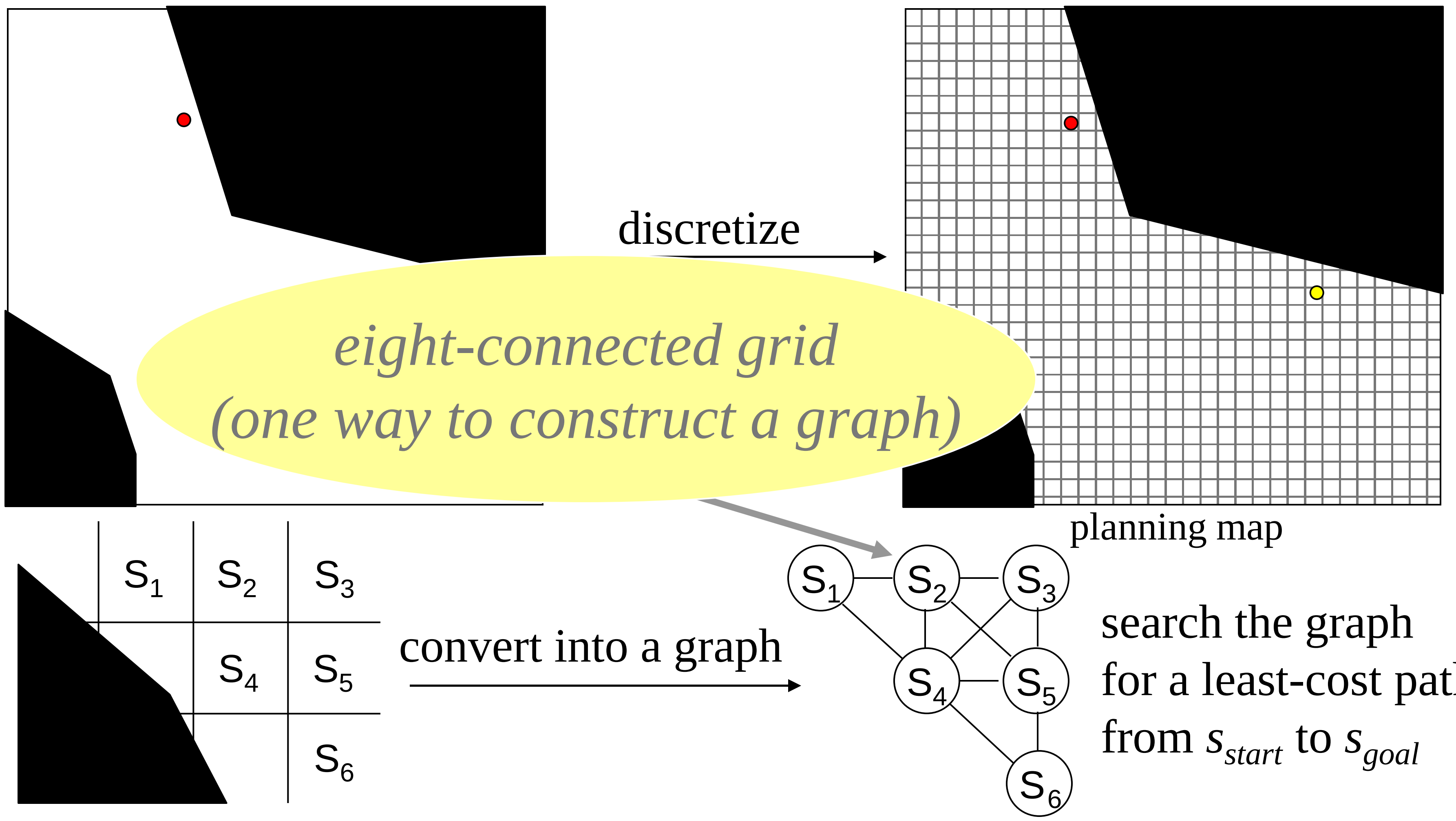
planning map



search the graph for a least-cost path from s_{start} to s_{goal}

Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path

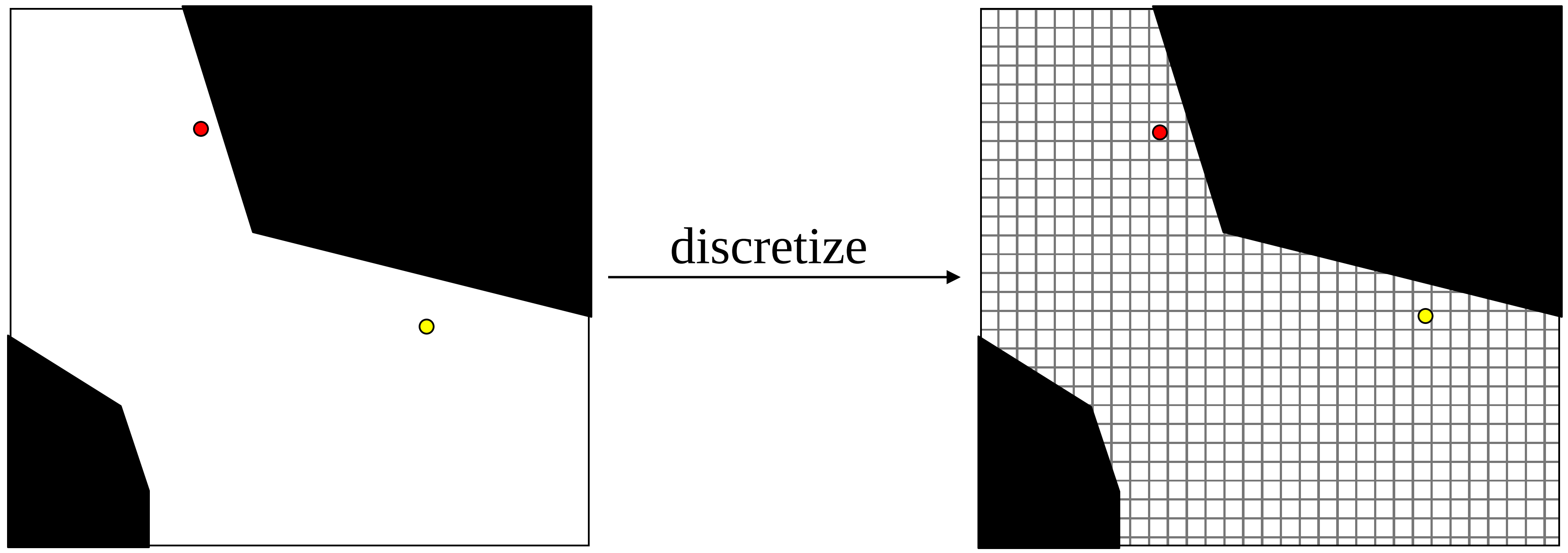

discretize
eight-connected grid
(one way to construct a graph)
planning map
S_1 S_2 S_3 S_4 S_5 S_6
convert into a graph
S_1 S_2 S_3 S_4 S_5 S_6

search the graph for a least-cost path from s_{start} to s_{goal}

Planning via Cell Decomposition

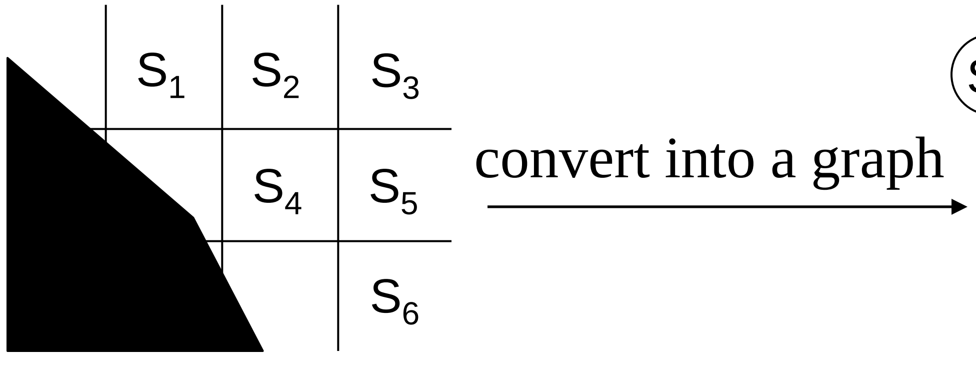
- Approximate Cell Decomposition:
  - construct a graph and search it for a least-cost path
    - VERY popular due to its simplicity
      - expensive in high-dimensional spaces

construct the grid on-the-fly, i.e. while planning – still expensive



Planning via Cell Decomposition

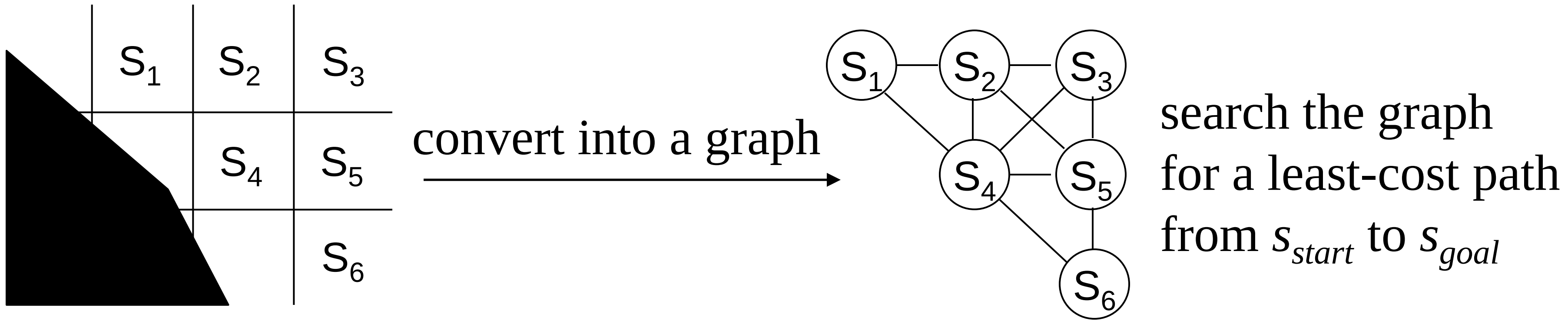
- Approximate Cell Decomposition:
  - what to do with partially blocked cells?



search the graph for a least-cost path from s_{start} to s_{goal}

Planning via Cell Decomposition

- Approximate Cell Decomposition:
    - what to do with partially blocked cells?
    - make it untraversable – incomplete (may not find a path that exists)

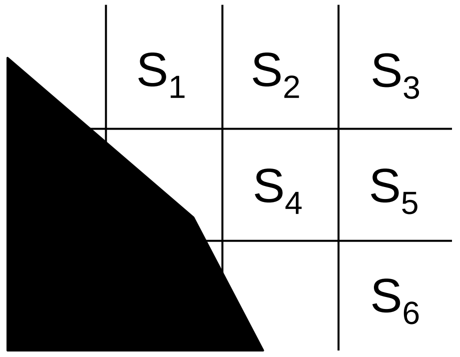


S_1
S_2
S_3
S_4
S_5
S_6
convert into a graph
search the graph for a least-cost path from s_{start} to s_{goal}

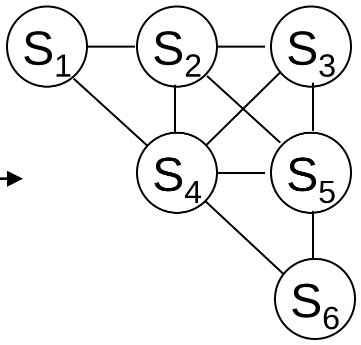
Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - what to do with partially blocked cells?
  - make it traversable – unsound (may return invalid path)

so, what's the solution?



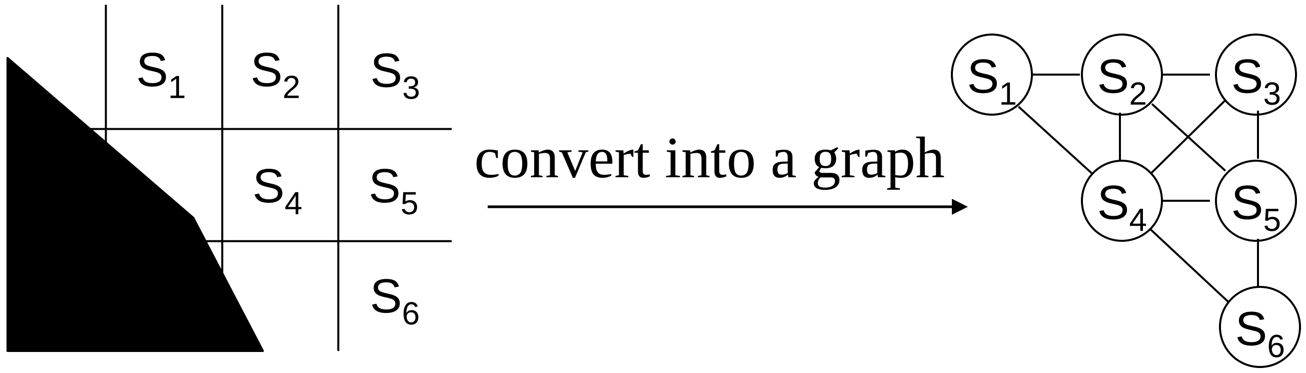
convert into a graph



search the graph for a least-cost path from s_{start} to s_{goal}

Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - solution 1:
    - make the discretization very fine
    - expensive, especially in high-D

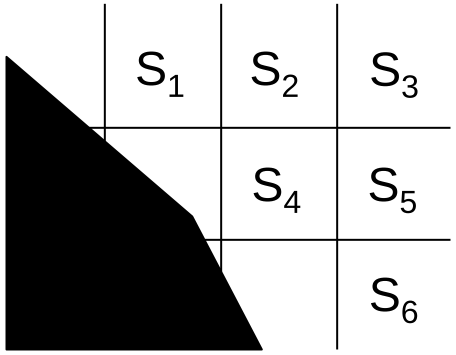


search the graph for a least-cost path from s_{start} to s_{goal}

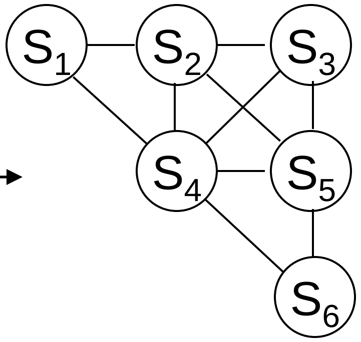
Planning via Cell Decomposition

- Approximate Cell Decomposition:
  - solution 2:
    - make the discretization adaptive
    - various ways possible

How?



convert into a graph $\longrightarrow$

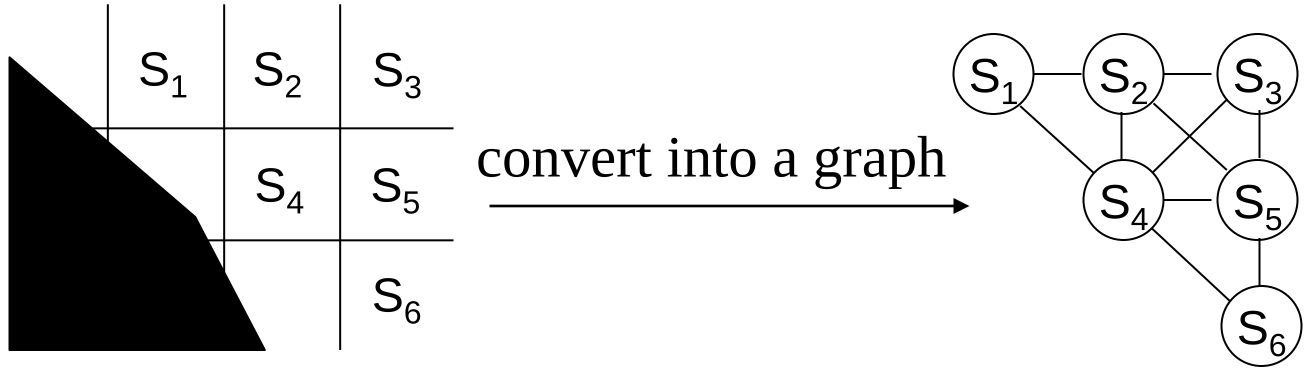


search the graph for a least-cost path from s_{start} to s_{goal}

Planning via Cell Decomposition

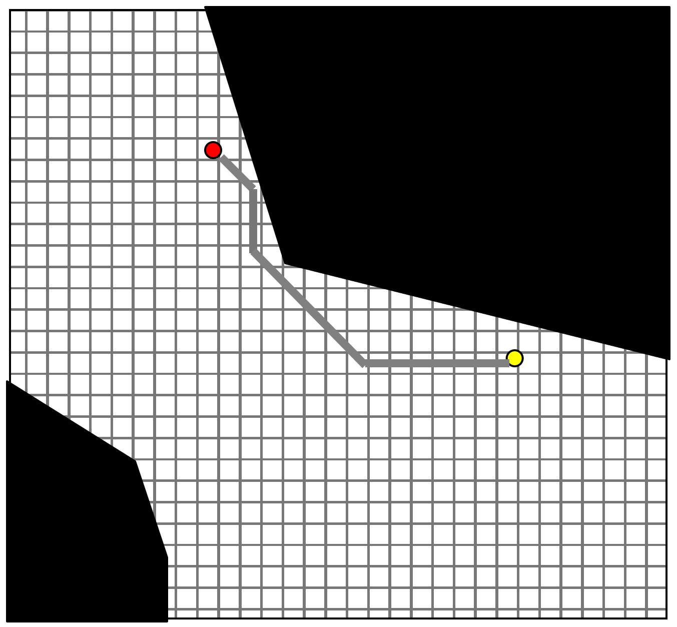
- Graph construction:
  - connect neighbors

eight-connected grid



Planning via Cell Decomposition

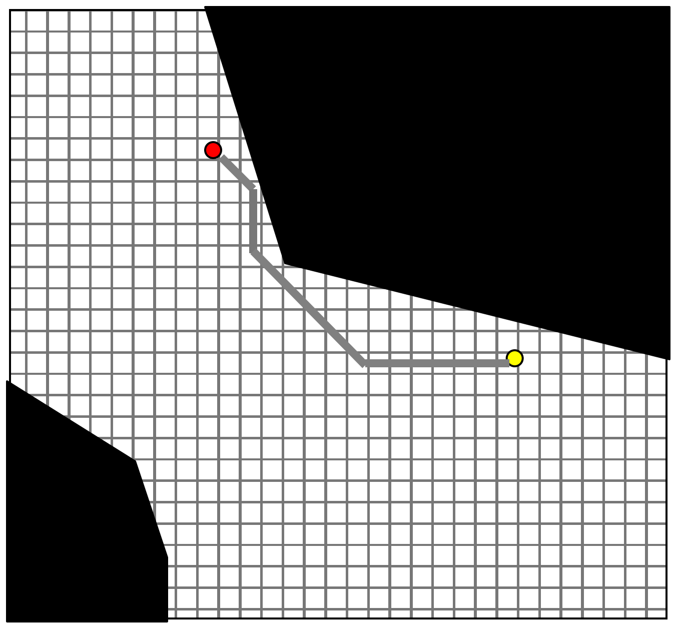
- Graph construction:
  - connect neighbors
  - path is restricted to 45^o degrees



Planning via Cell Decomposition

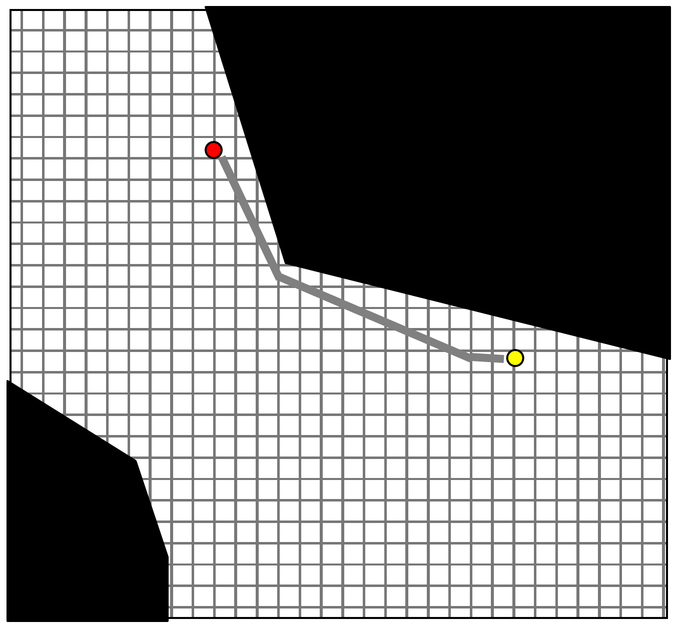
- Graph construction:
  - connect neighbors
  - path is restricted to 45^{o} degrees

Will planning in 3D help?

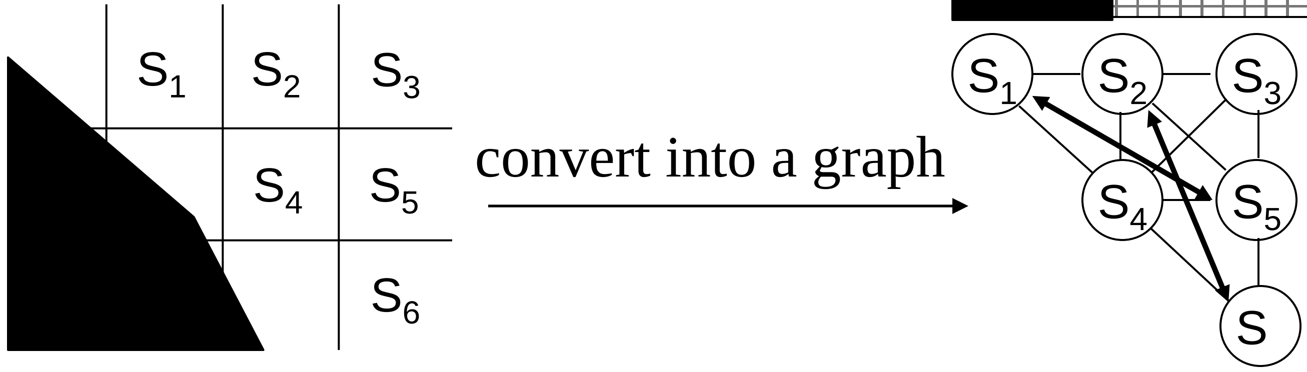


Planning via Cell Decomposition

- Graph construction:
  - connect cells to neighbor of neighbors
  - path is restricted to 22.5º degrees



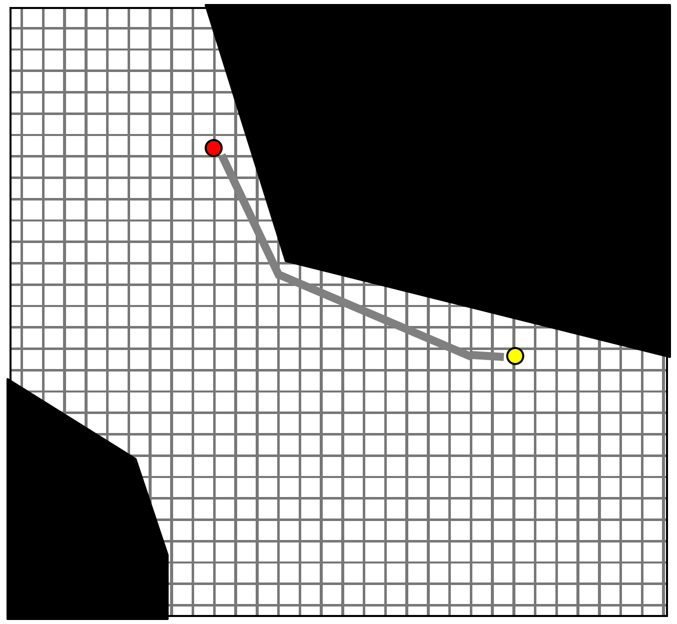
16-connected grid



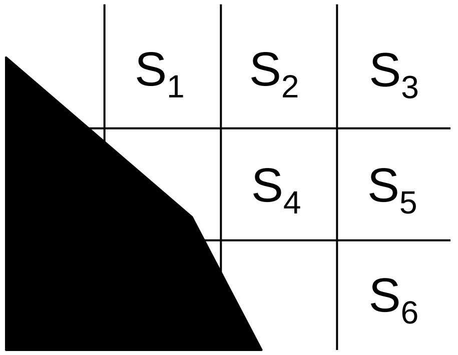
Planning via Cell Decomposition

- Graph construction:
  - connect cells to neighbor of neighbors
  - path is restricted to 22.5° degrees

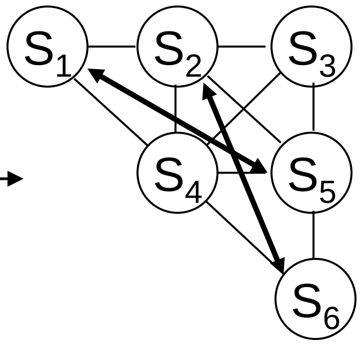
Disadvantages?



16-connected grid



convert into a graph



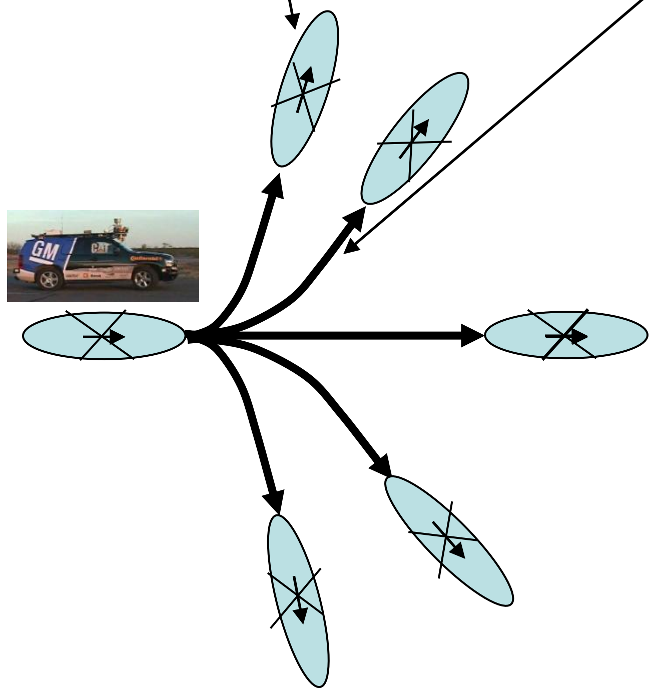
Planning via Cell Decomposition

- Graph construction:
  - lattice graph for computing smooth (realistic) paths

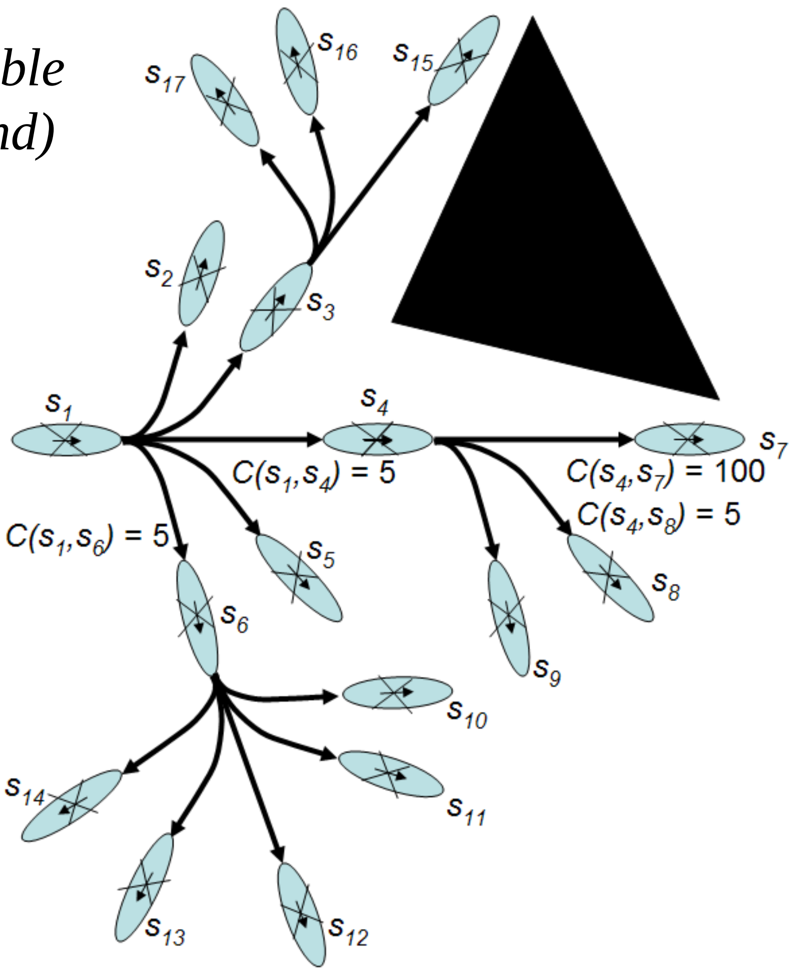
outcome state is the center of the corresponding cell

each transition is feasible (constructed beforehand)

action template



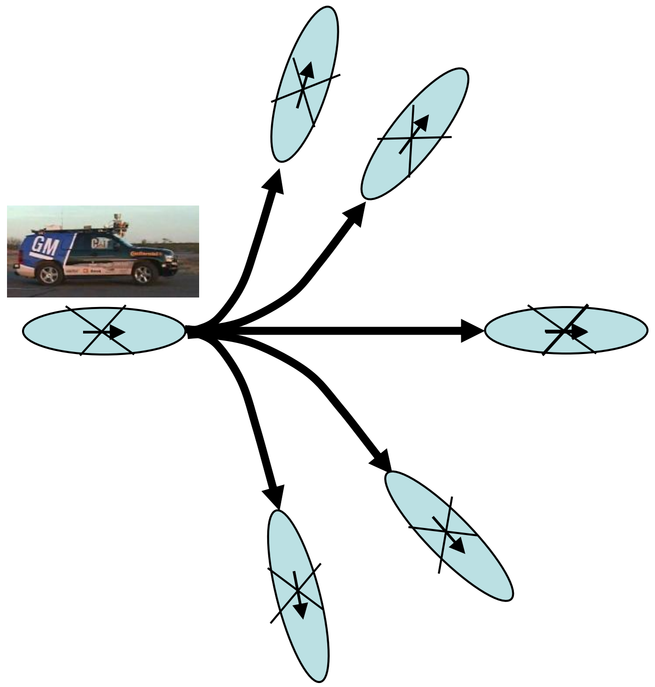
replicate it $\xrightarrow{\hspace{3em}}$ *online*



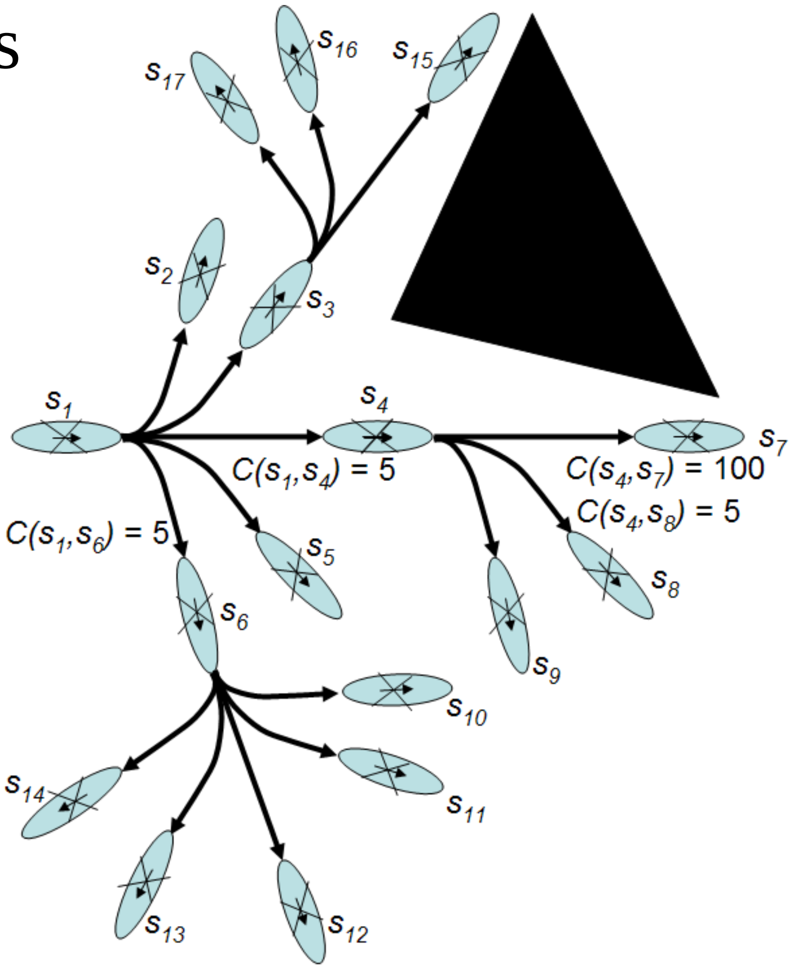
Planning via Cell Decomposition

- Graph construction:
  - lattice graph
  - pros: sparse graph, feasible paths
  - cons: possible incompleteness

action template

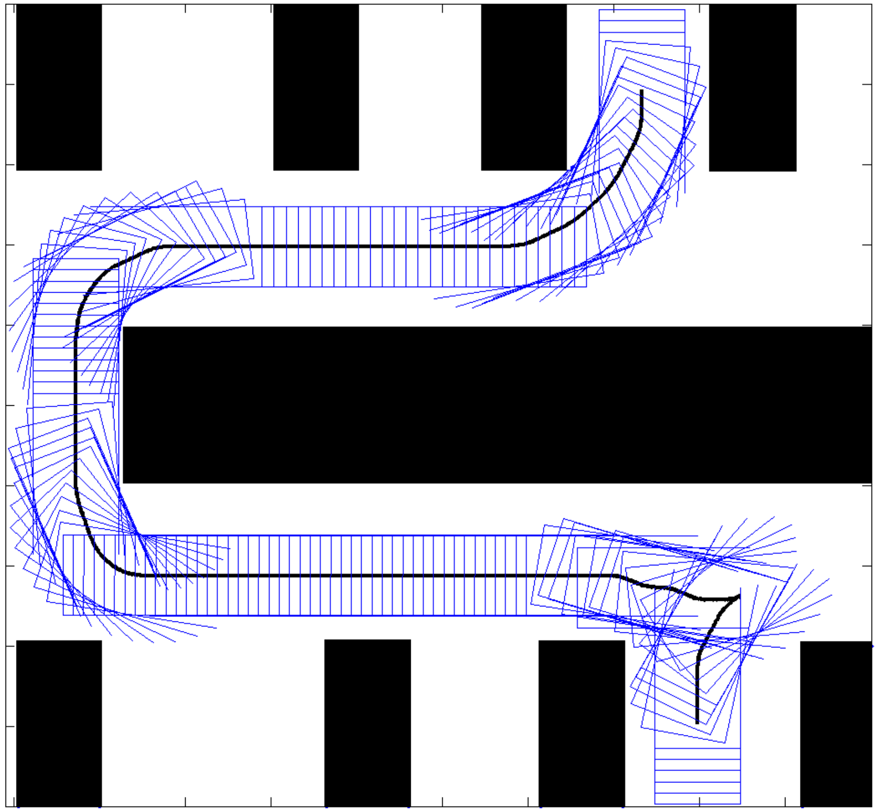


replicate it $\xrightarrow{\qquad}$ *online*



Planning via Cell Decomposition

- Graph construction:
    - lattice graph



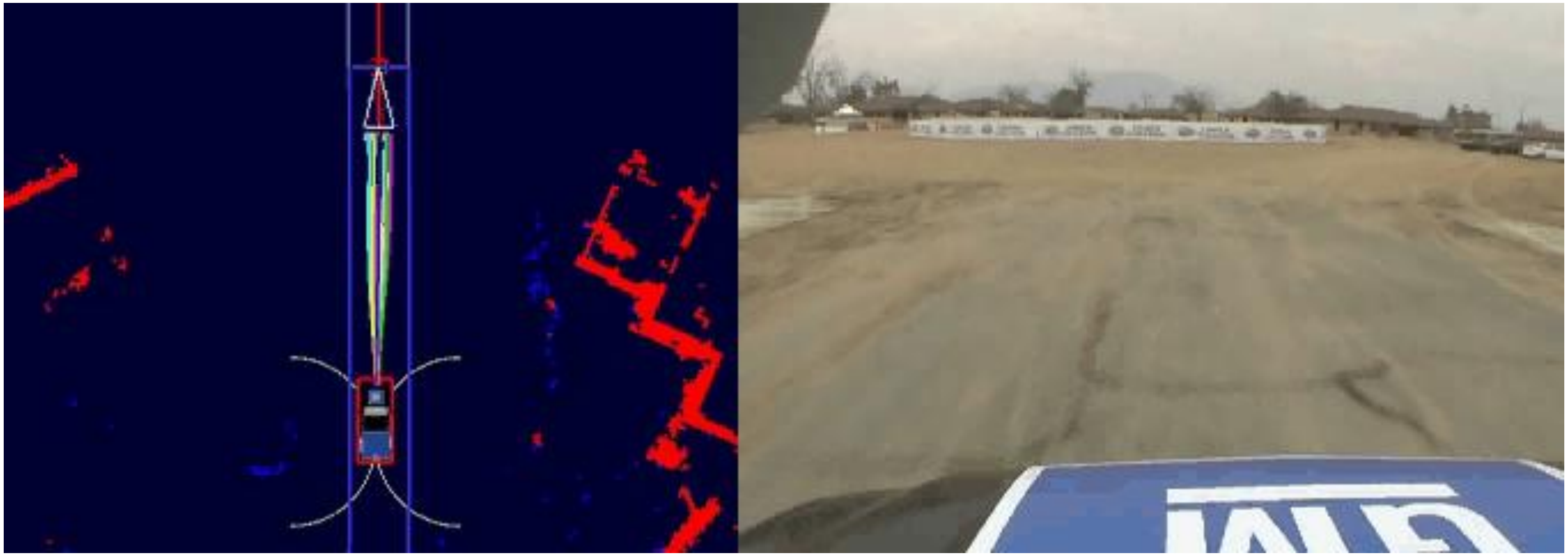
Planning via Cell Decomposition

- Graph construction:
  - lattice graph

planning on 4D lattice graph:

each state represents <x,y,orientation,velocity>

each edge represents a short feasible motion between corresponding cells



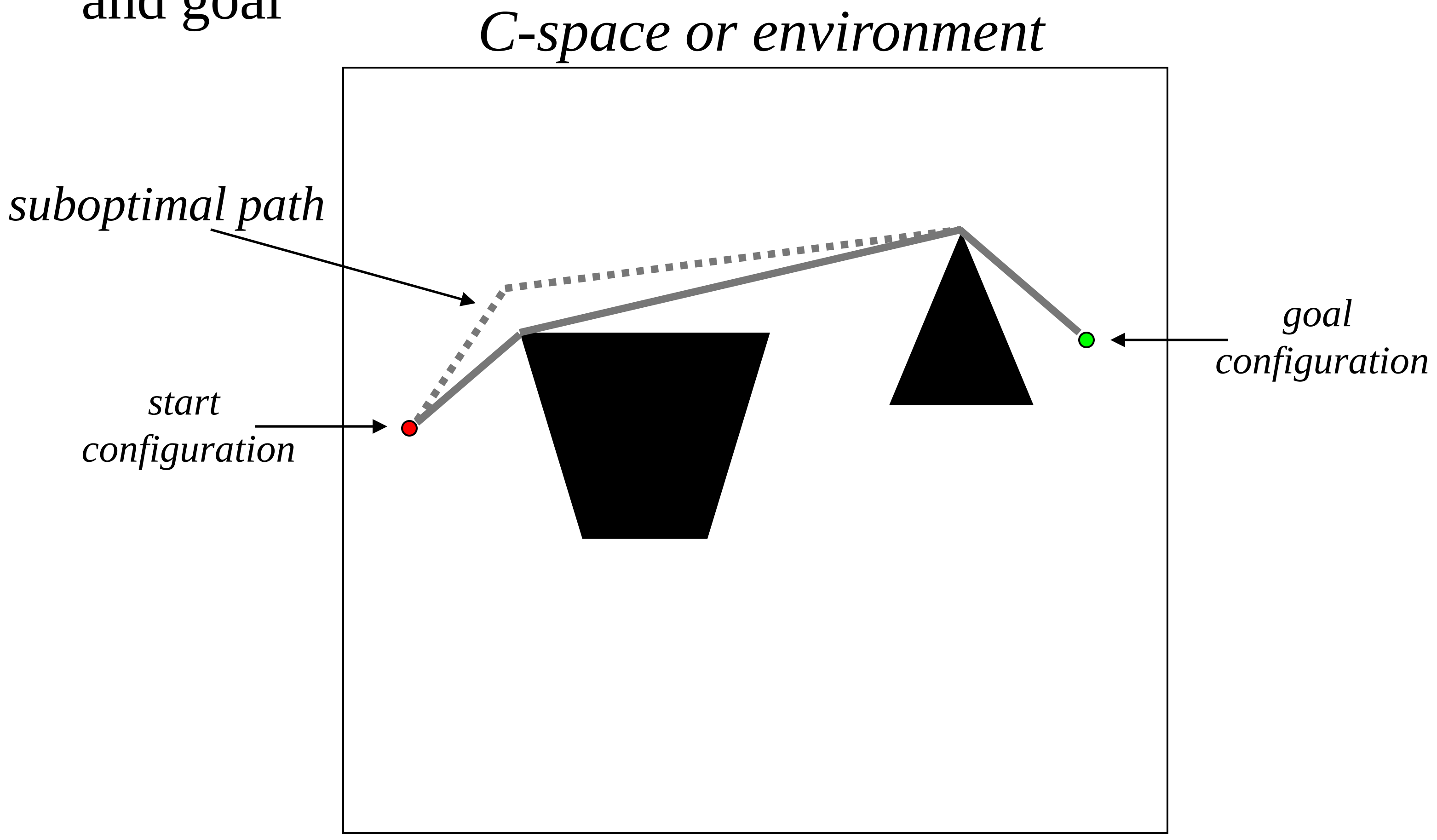
Skeletonization of the C-Space

Skeletonization: construction of a unidimensional representation of the C-space

- Visibility graph
- Voronoi diagram
- Probabilistic road-map
- Navmeshes

Planning via Skeletonization

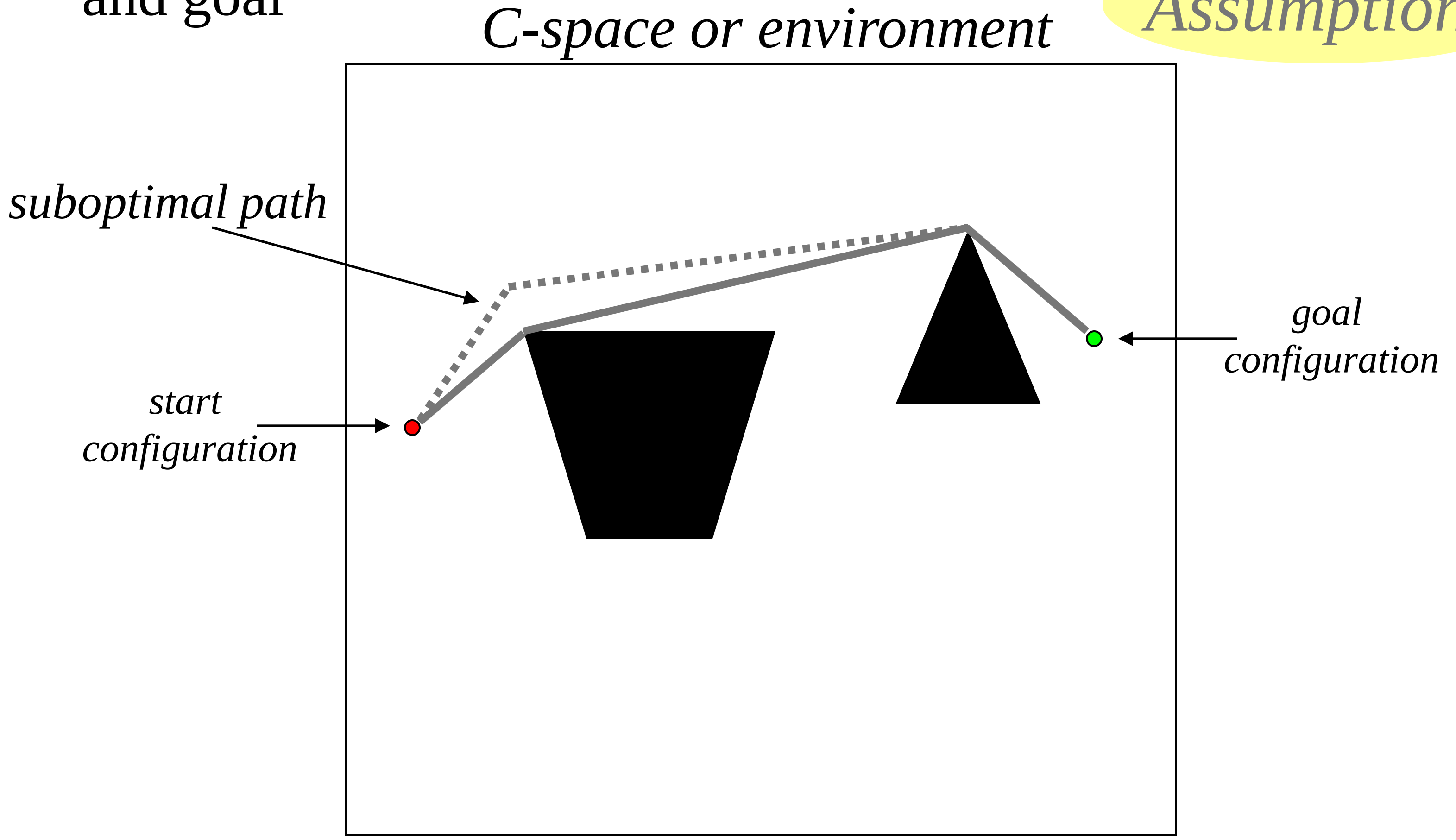
- Visibility Graphs [Wesley & Lozano-Perez '79]
  - based on idea that the shortest path consists of obstacle-free straight line segments connecting all obstacle vertices and start and goal



Planning via Skeletonization

- Visibility Graphs
  - based on idea that the shortest path consists of obstacle-free straight line segments connecting all obstacle vertices and start and goal

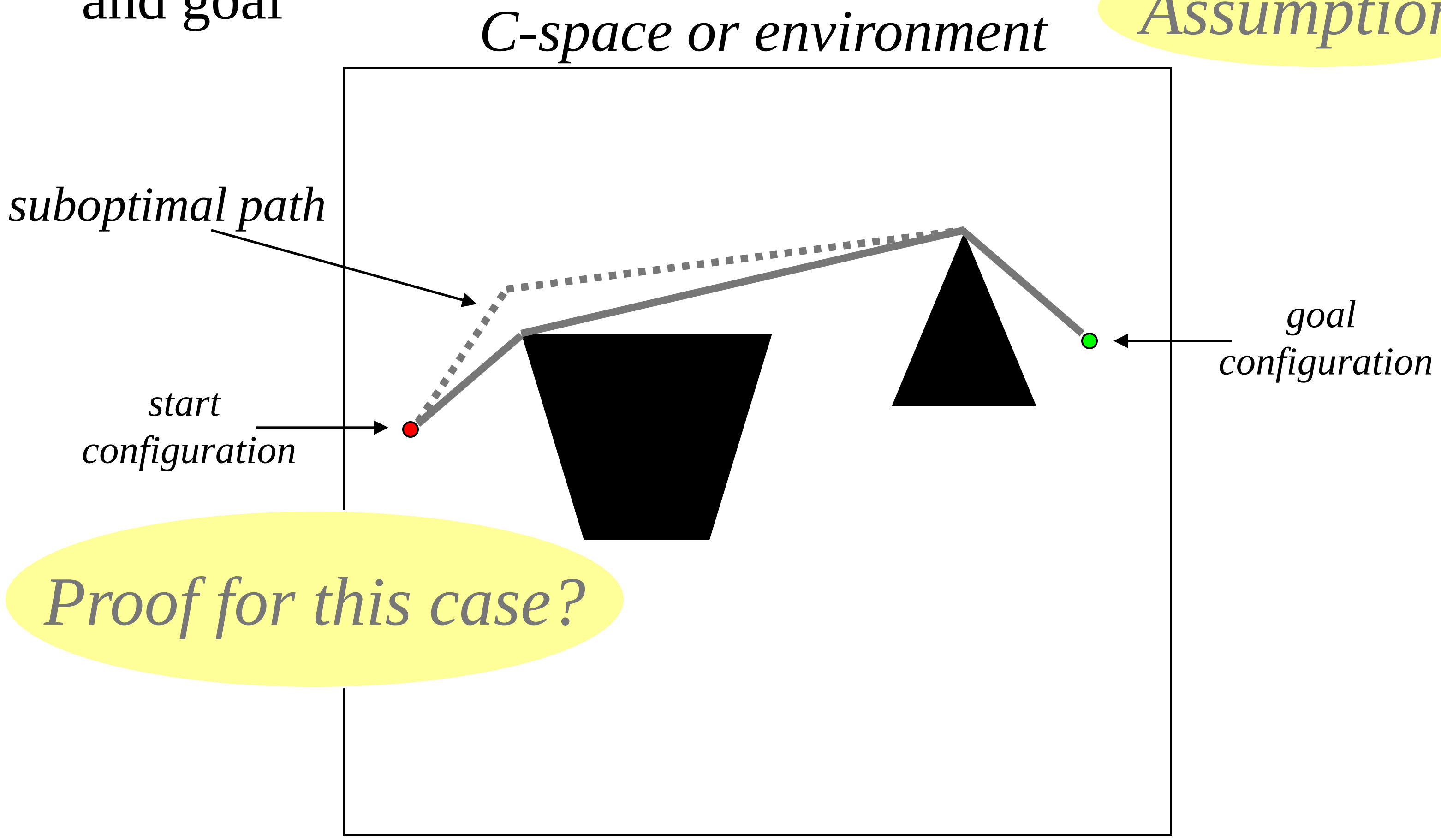
Assumption?



Planning via Skeletonization

- Visibility Graphs
  - based on idea that the shortest path consists of obstacle-free straight line segments connecting all obstacle vertices and start and goal

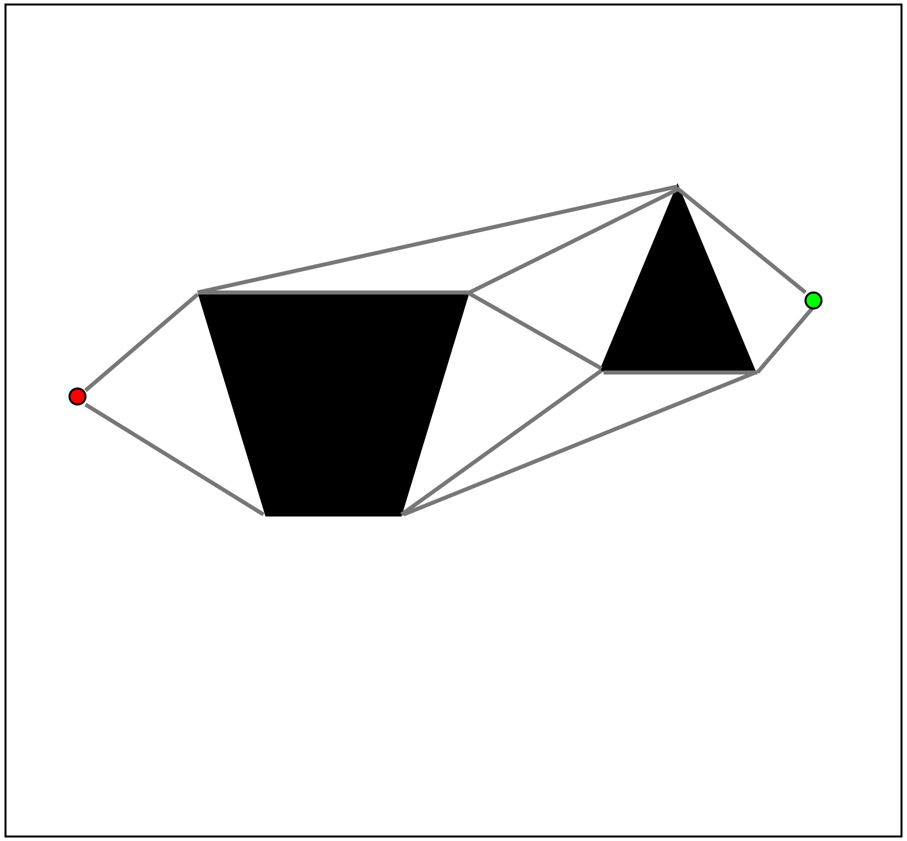
Assumption?



Proof for this case?

Planning via Skeletonization

- Visibility Graphs [Wesley & Lozano-Perez '79]
  - construct a graph by connecting all vertices, start and goal by obstacle-free straight line segments (graph is $O(n^2)$, where n - # of vert.)
  - search the graph for a shortest path



Planning via Skeletonization

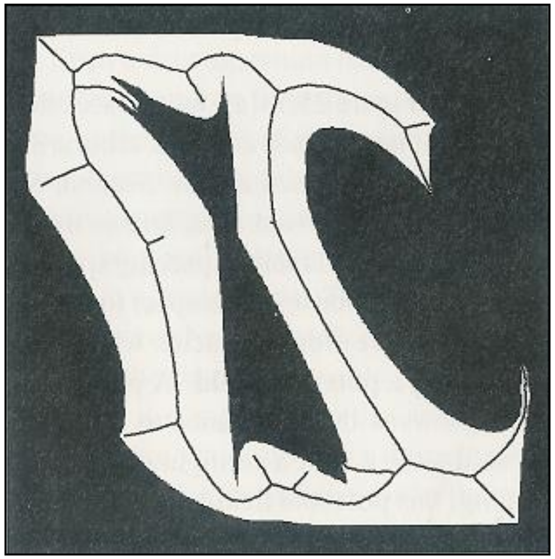
- Visibility Graphs
  - advantages:
    - independent of the size of the environment
  - disadvantages:
    - path is too close to obstacles
    - hard to deal with non-uniform cost function
    - hard to deal with non-polygonal obstacles

Planning via Skeletonization

- Voronoi diagrams [Rowat '79]

  - voronoi diagram: set of all points that are equidistant to two nearest obstacles

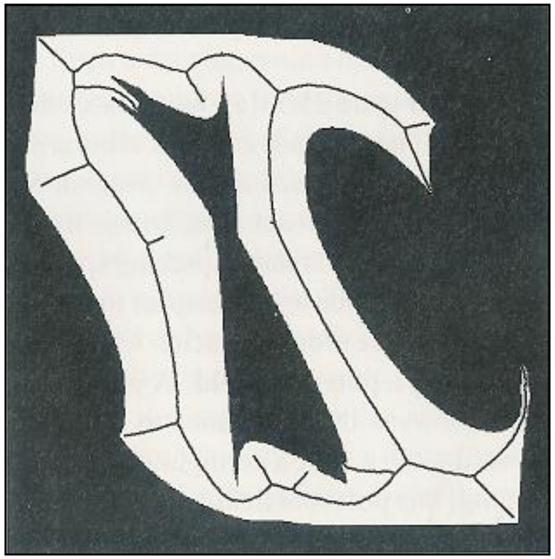
  - based on the idea of maximizing clearance instead of minimizing travel distance



the example above is borrowed from "AI: A Modern Approach" by RusselL & P. Norvig

Planning via Skeletonization

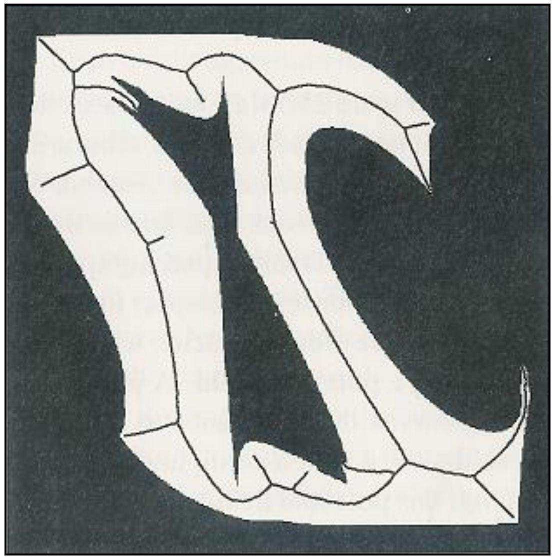
- Voronoi diagrams
  - compute voronoi diagram (O (n log n), where n - # of invalid configurations)
  - add a shortest path segment from start to the nearest segment of voronoi diagram
  - add a shortest path segment from goal to the nearest segment of voronoi diagram
  - compute shortest path in the graph



the example above is borrowed from "AI: A Modern Approach" by RusselL & P. Norvig

Planning via Skeletonization

- Voronoi diagrams
  - advantages:
    - tends to stay away from obstacles
    - independent of the size of the environment
  - disadvantages:
    - can result in highly suboptimal paths

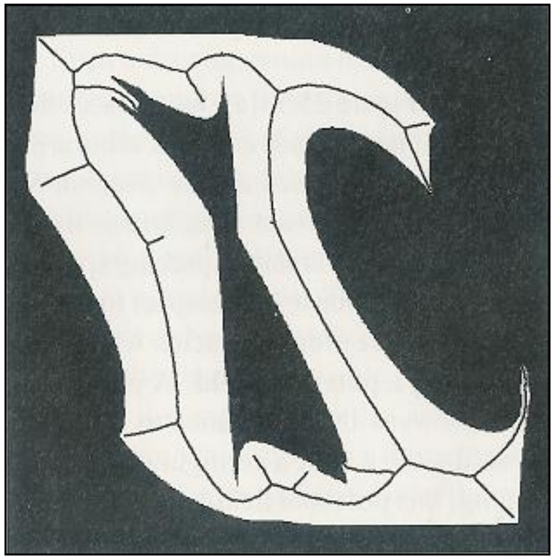


the example above is borrowed from "AI: A Modern Approach" by RussellL & P. Norvig

Planning via Skeletonization

- Voronoi diagrams
  - advantages:
    - tends to stay away from obstacles
    - independent of the size of the environment
  - disadvantages:
    - can result in highly suboptimal p

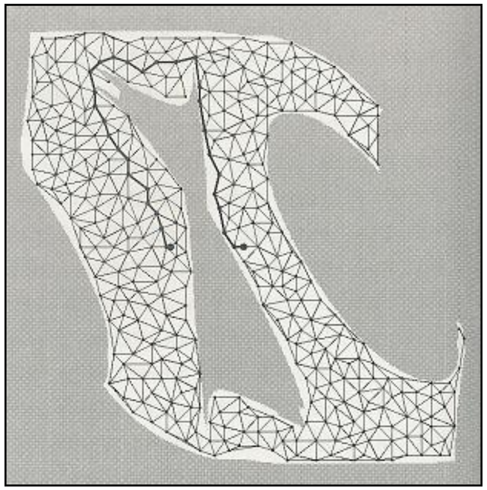
In which environments?



the example above is borrowed from "AI: A Modern Approach" by RussellL & P. Norvig

Planning via Skeletonization

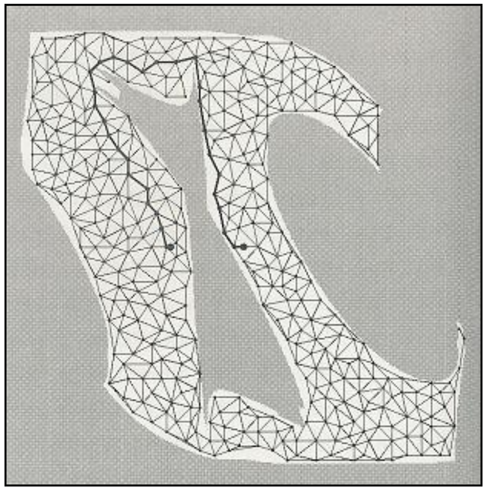
- Probabilistic roadmaps [Kavraki et al. '96]
  - construct a graph by:
    - randomly sampling valid configurations
    - adding edges in between the samples that are easy to connect with a straight line
  - add start and goal configurations to the graph with appropriate edges
  - compute shortest path in the graph



the example above is borrowed from "AI: A Modern Approach" by RusselL & P. Norvig

Planning via Skeletonization

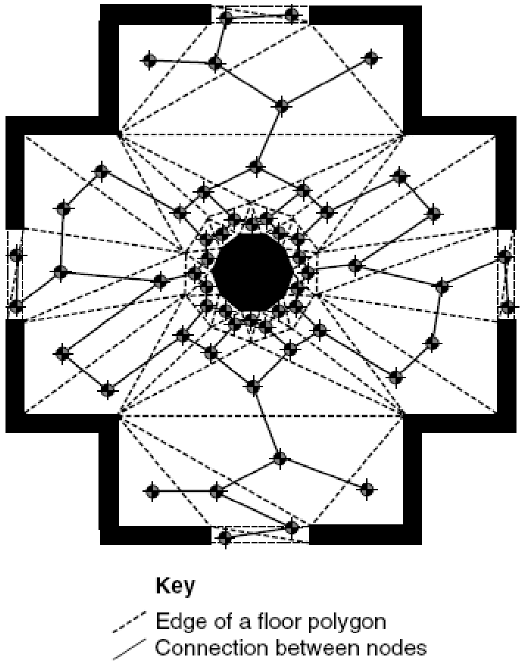
- Probabilistic roadmaps [Kavraki et al. '96]
  - simple and highly effective (especially in >2D)
  - very popular
  - can result in suboptimal paths, no guarantees on suboptimality
  - difficulty with narrow passages



the example above is borrowed from "AI: A Modern Approach" by RusselL & P. Norvig

Planning via Skeletonization

- **Navmeshes**
  - pick centers of triangles defining floor plan as graph vertices
  - semi-manual but very popular in games
  - can result in suboptimal paths, no guarantees on suboptimality

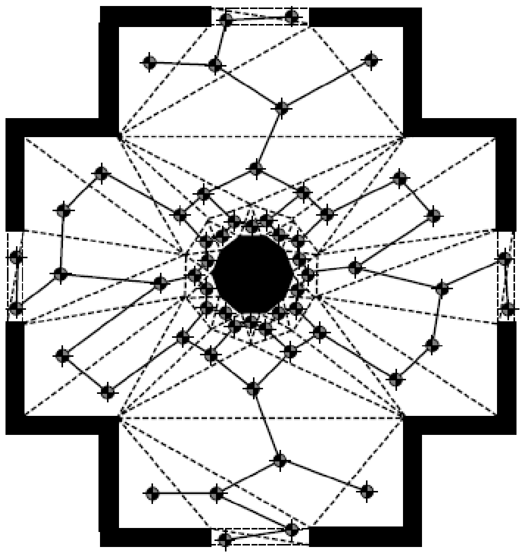


from “Artificial Intelligence for Games” by I. Millington & J. y ge

Planning via Skeletonization

- **Navmeshes**
  - pick centers of triangles defining floor plan as graph vertices
  - semi-manual but very popular in games
  - can result in suboptimal paths, no guarantees on suboptimality

Other disadvantages?



Key

┄ Edge of a floor polygon

— Connection between nodes

from "Artificial Intelligence for Games" by I. Millington & J. y ge

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

Searching Graphs for a Least-cost Path

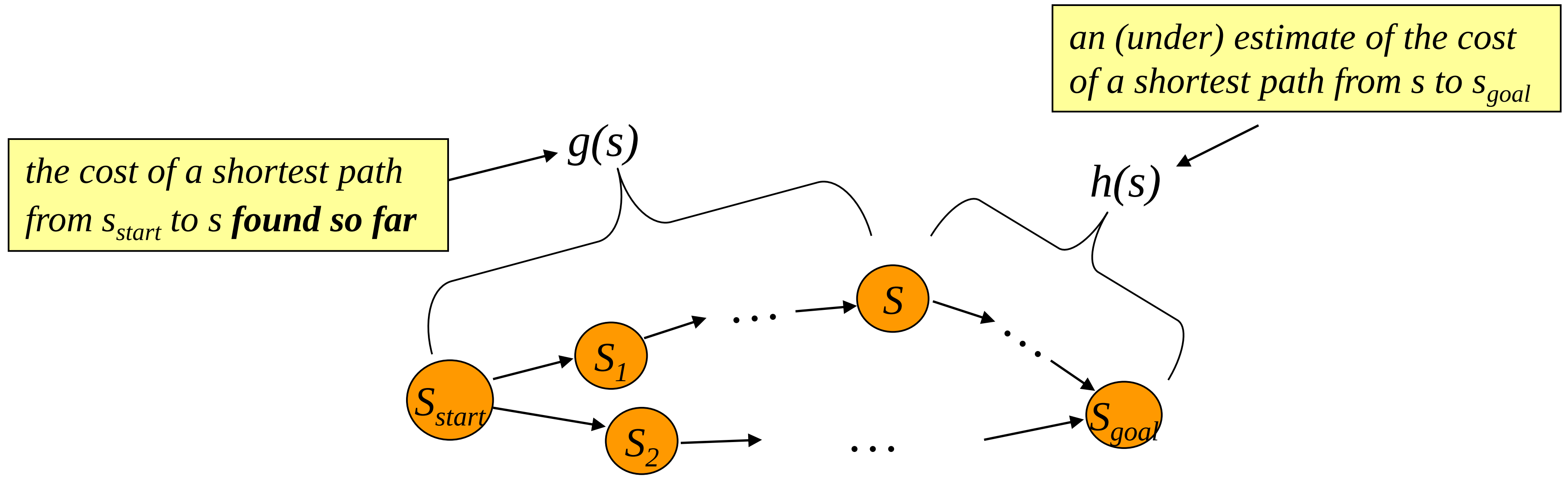
- Once a graph is constructed (from skeletonization or uniform cell decomposition or adaptive cell decomposition or lattice or whatever else), we need to search it for a least-cost path

$$\begin{tikzcd}
& S_2 \arrow[r, "2"] \arrow[dd, "1"] & S_1 \arrow[dr, "2"] & \\
S_{start} \arrow[ur, "1"] & & & S_{goal} \\
& S_4 \arrow[r, "3"] & S_3 \arrow[ur, "1"] &
\end{tikzcd}$$

A* Search

- Computes optimal g-values for relevant states

at any point of time:

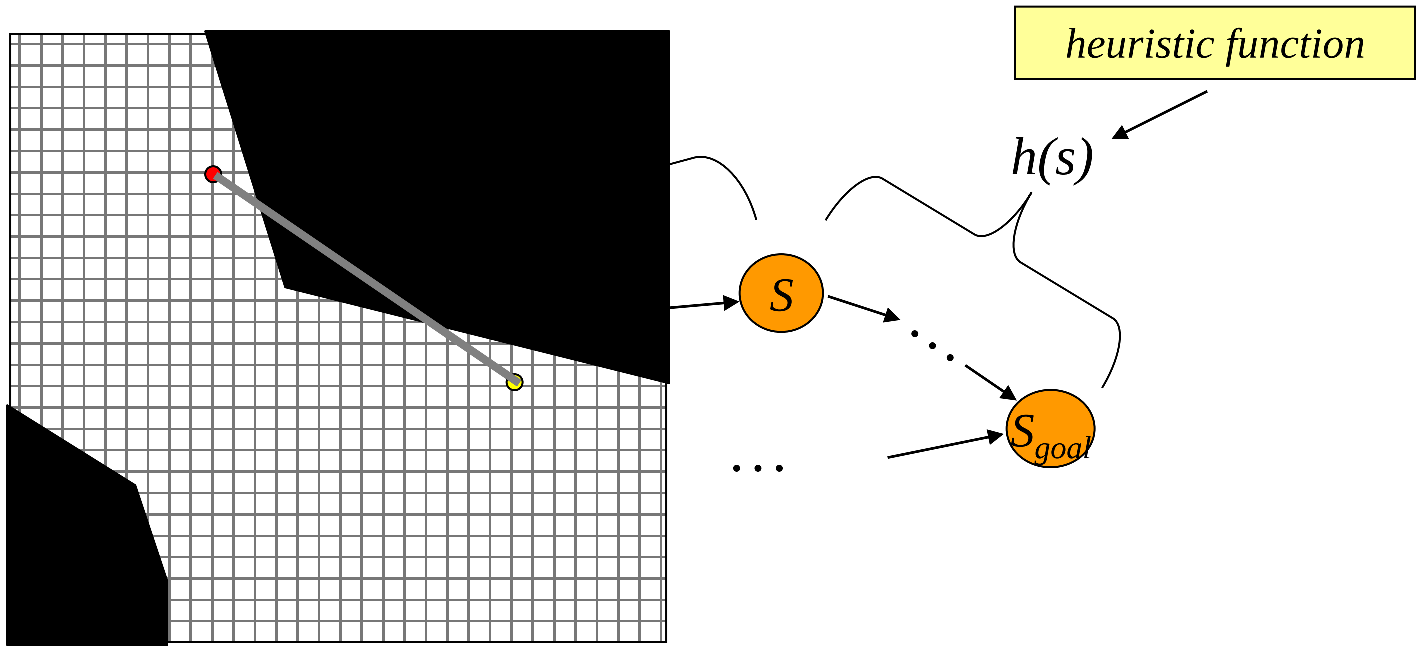


the cost of a shortest path from s_{start} to s ***found so far***
an (under) estimate of the cost of a shortest path from s to s_{goal}
$g(s)$
$h(s)$
S_{start}
S_1
S_2
S
S_{goal}

A* Search

- Computes optimal g-values for relevant states

at any point of time:



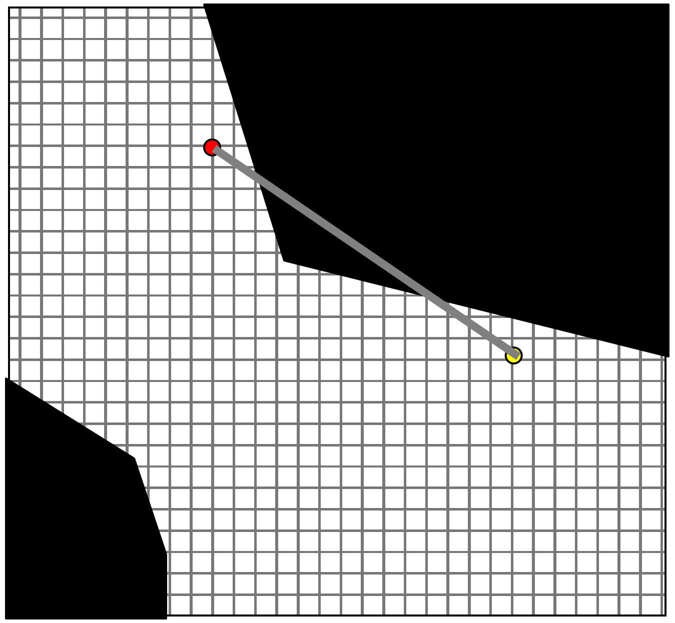
one popular heuristic function – Euclidean distance

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

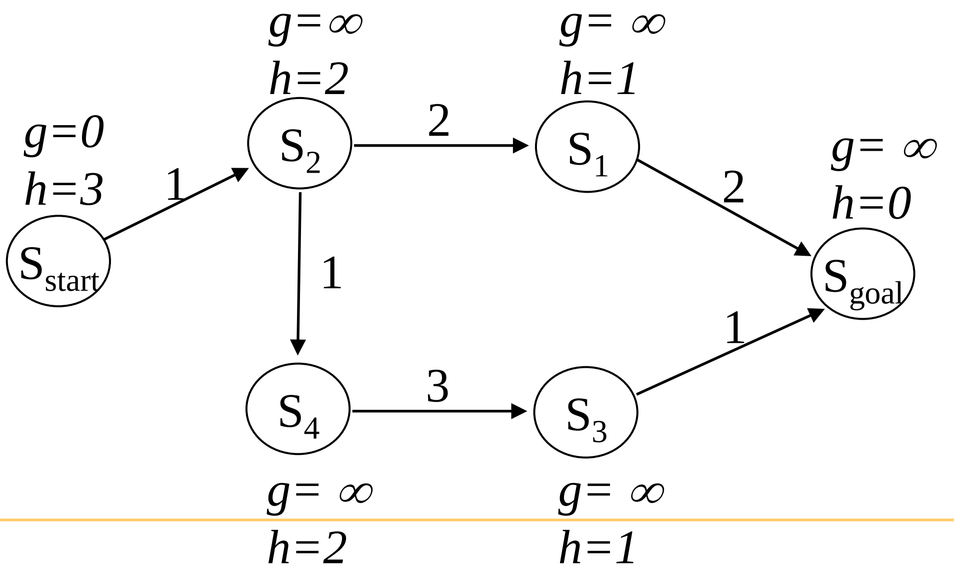
```
$g(s_{start}) = 0$; all other $g$-values are infinite; $OPEN = \{s_{start}\}$;
ComputePath();
publish solution;
```

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)

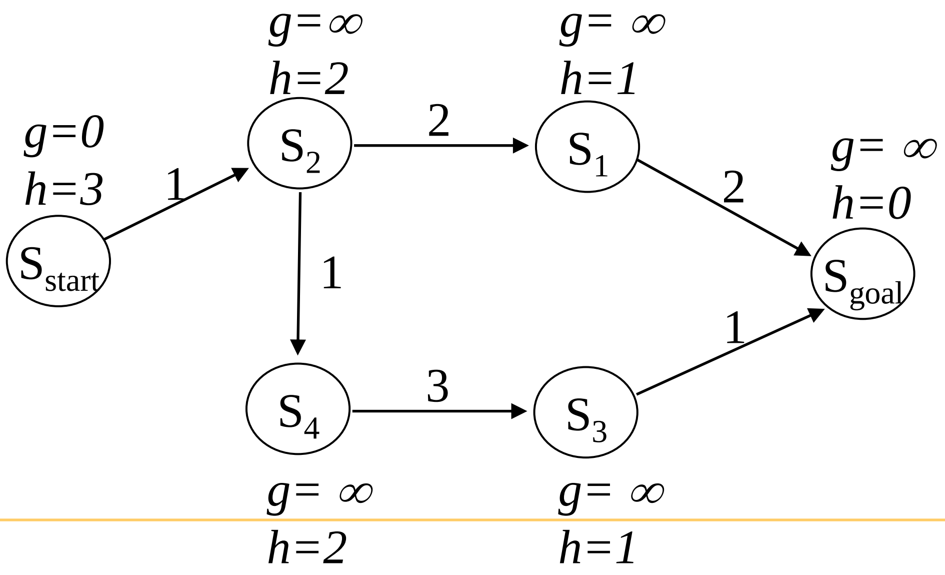


A* Search

- 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

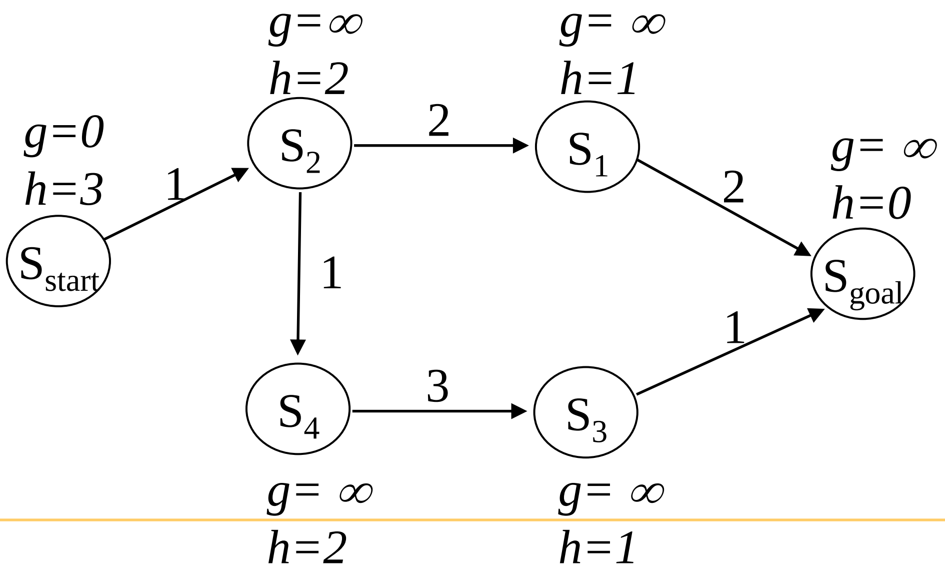
- Computes optimal g-values for relevant states

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

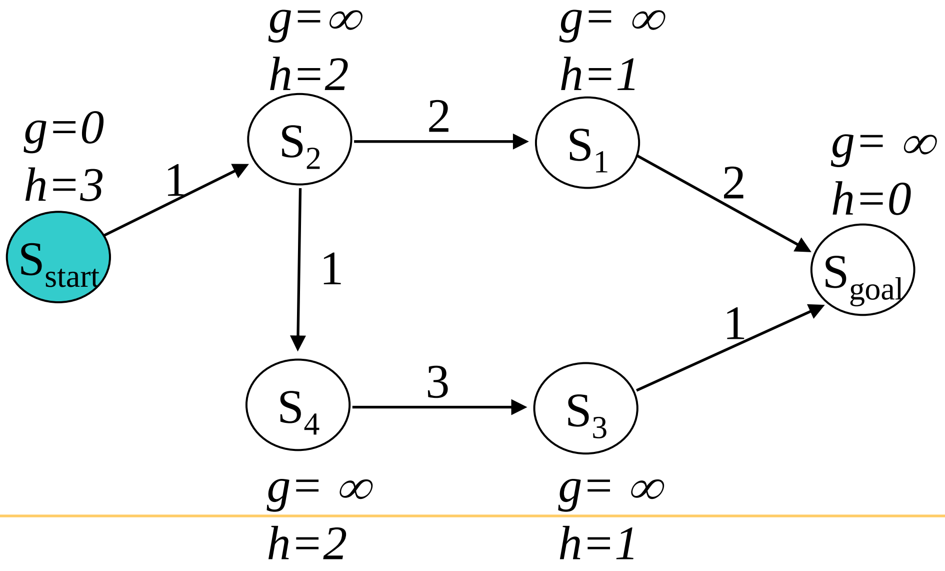


A* Search

- 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

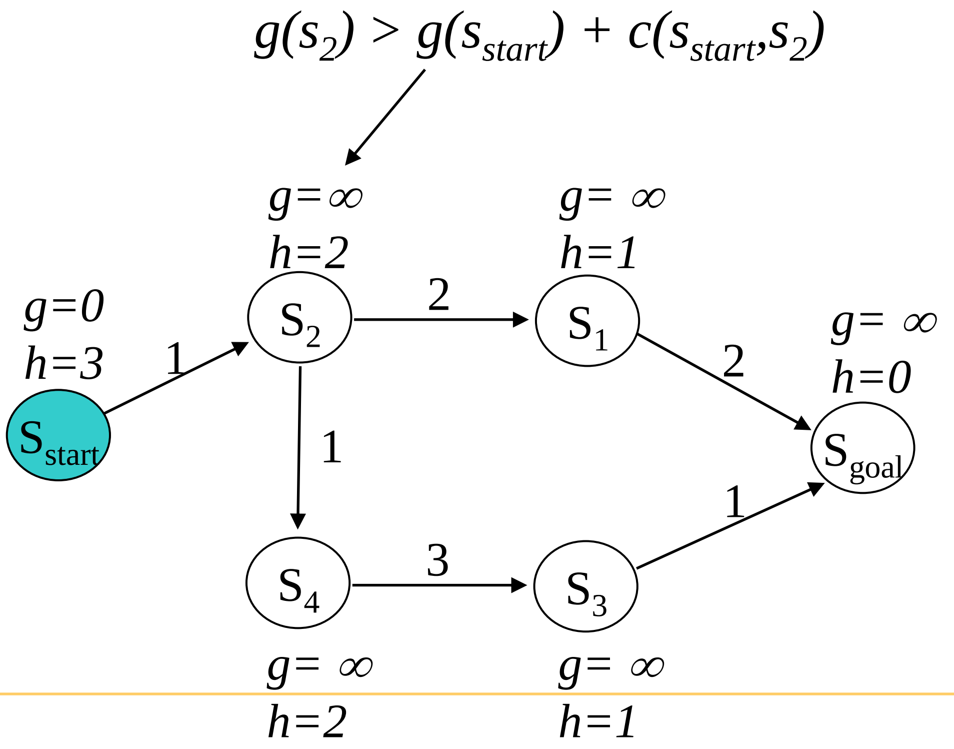
ComputePath function

```
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}

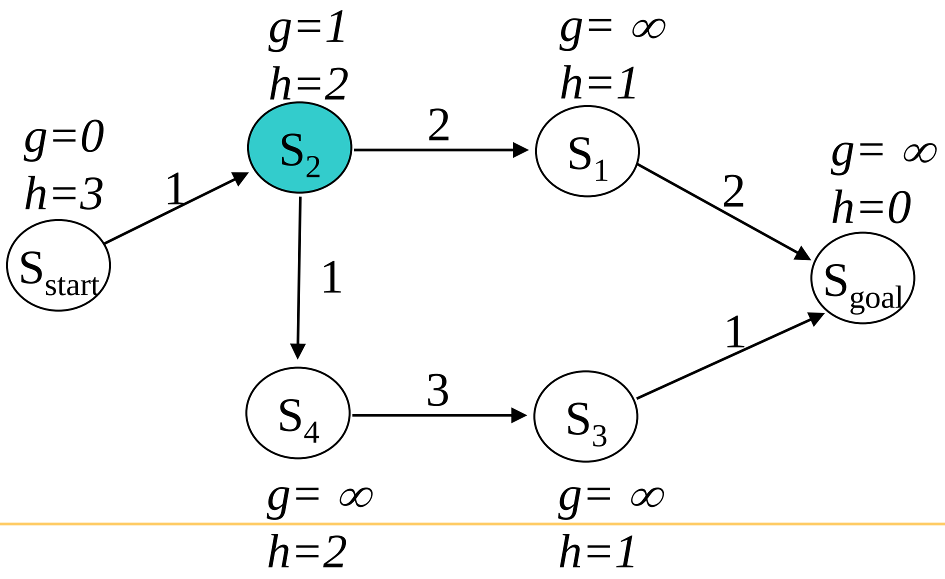


A* Search

- 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 $OPEN$;
```

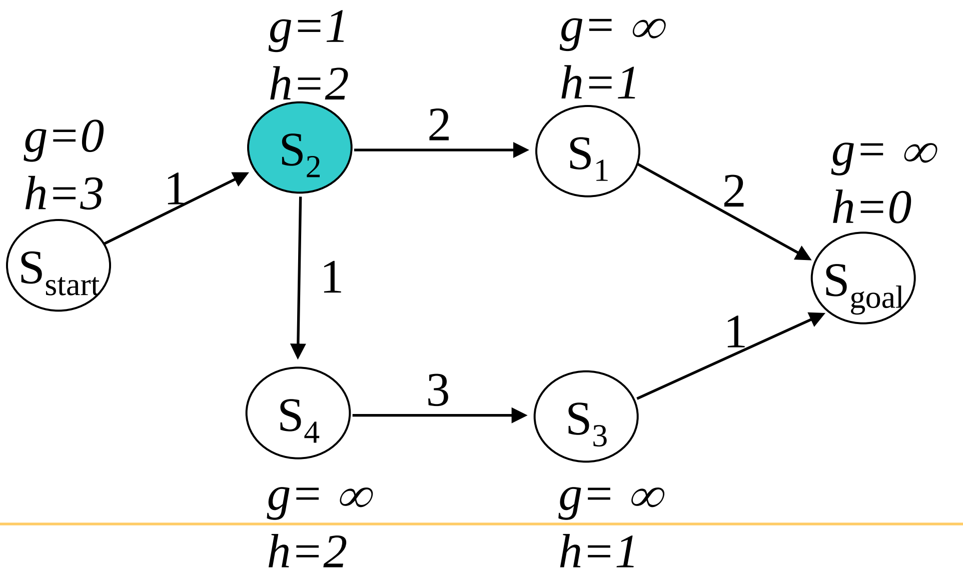


A* Search

- 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

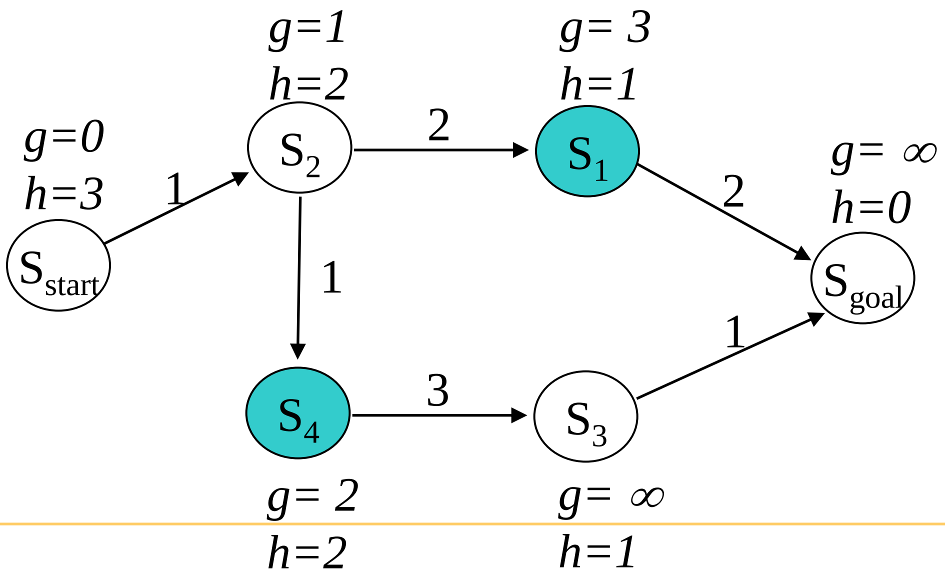
ComputePath function

```
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

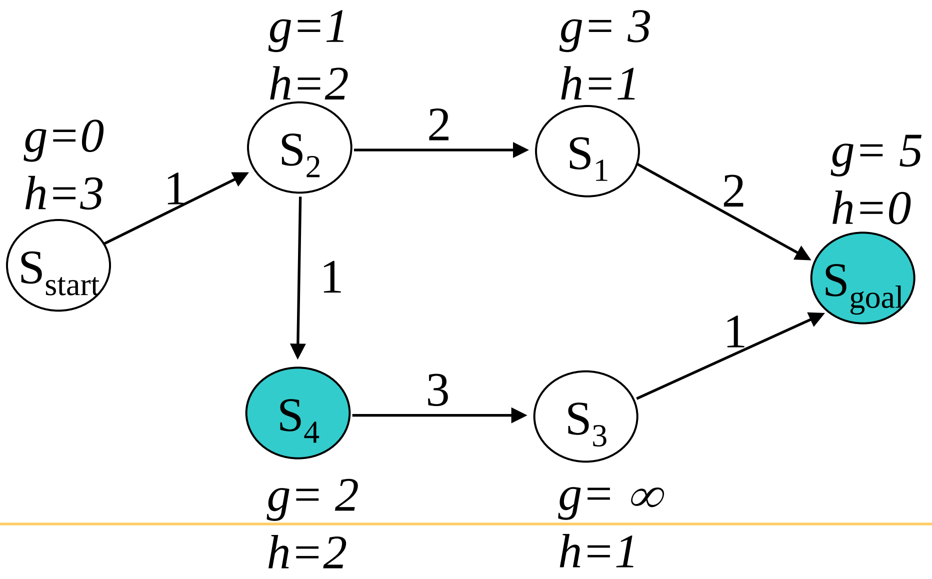
ComputePath function

```
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

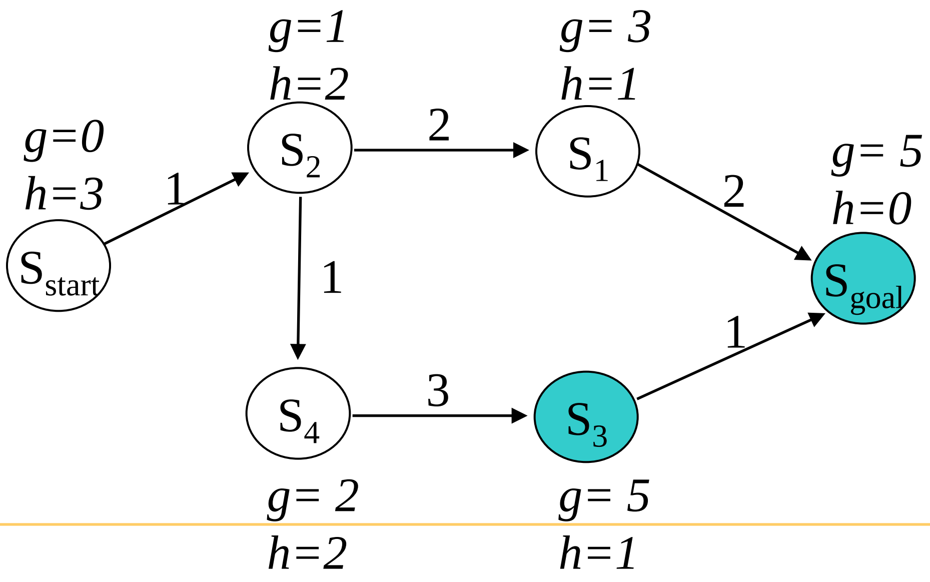
ComputePath function

```
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

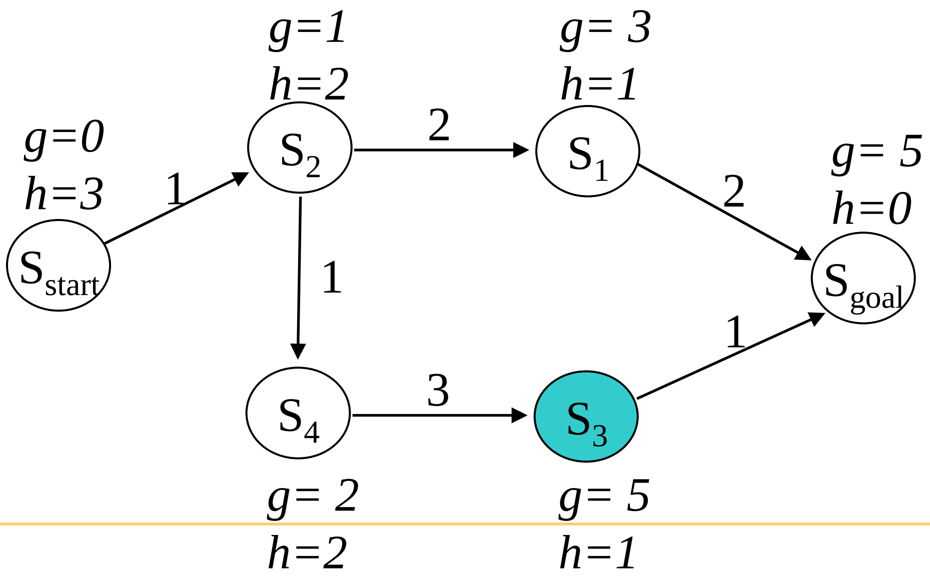
ComputePath function

```
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

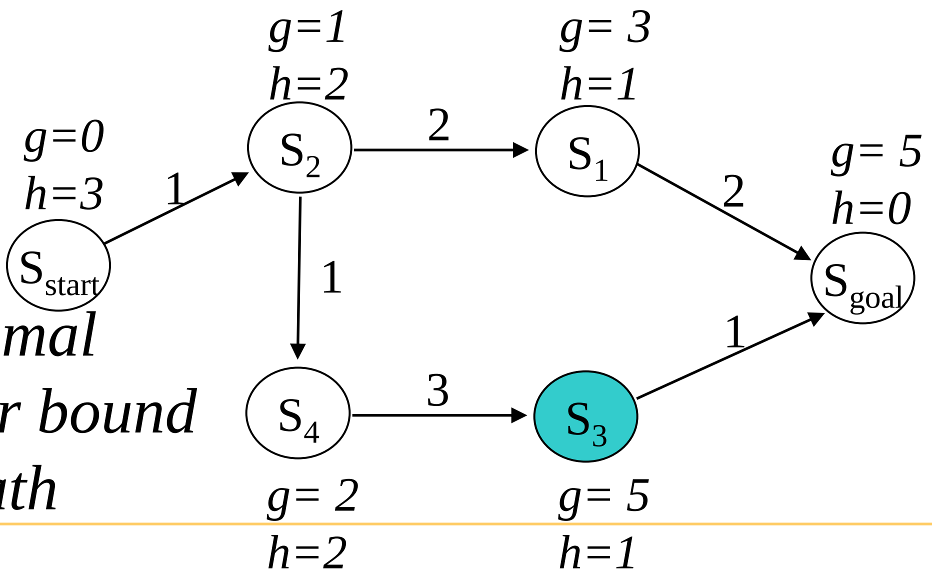


A* Search

- 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 $OPEN$;
```



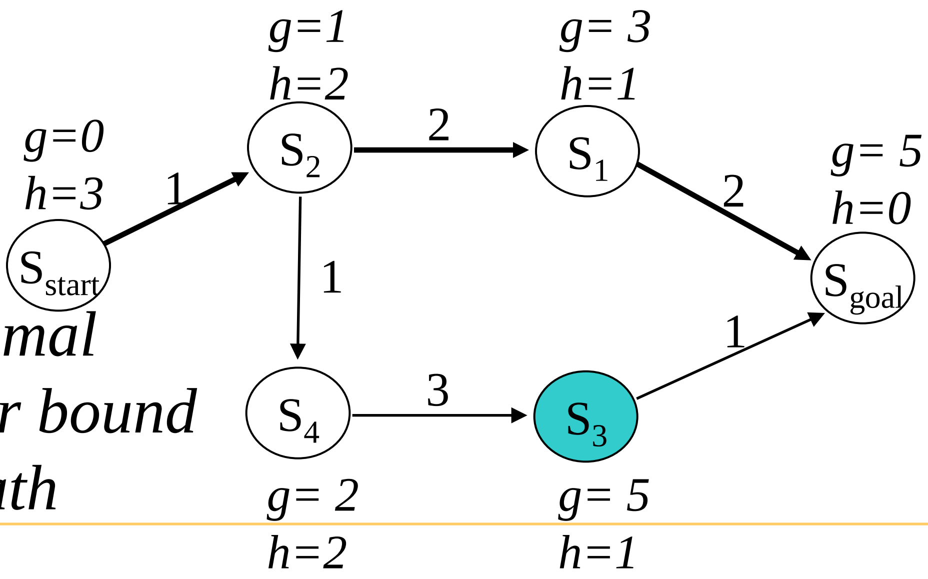
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

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 $OPEN$;
```



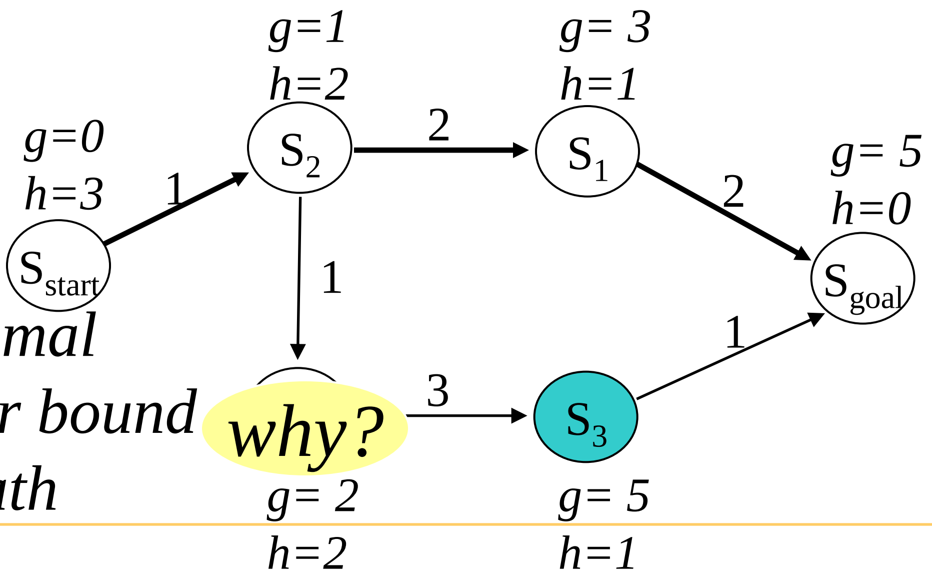
for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

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 $OPEN$;
```

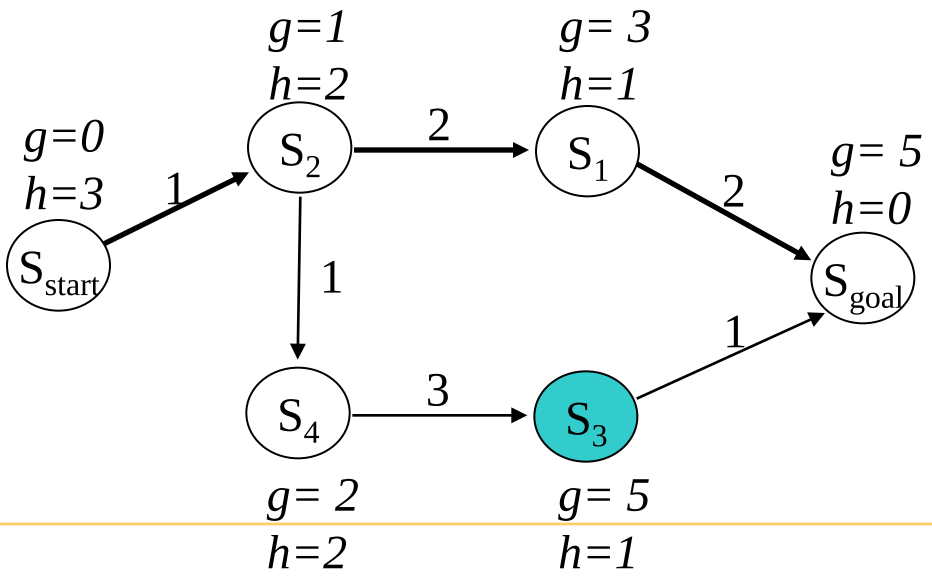


for every expanded state g(s) is optimal
for every other state g(s) is an upper bound
we can now compute a least-cost path

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



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 $g(s') > g(s) + c(s,s')$                                     > expansion of s
            $g(s') = g(s) + c(s,s')$;                                   |
            insert $s'$ into $OPEN$;                                    /
```

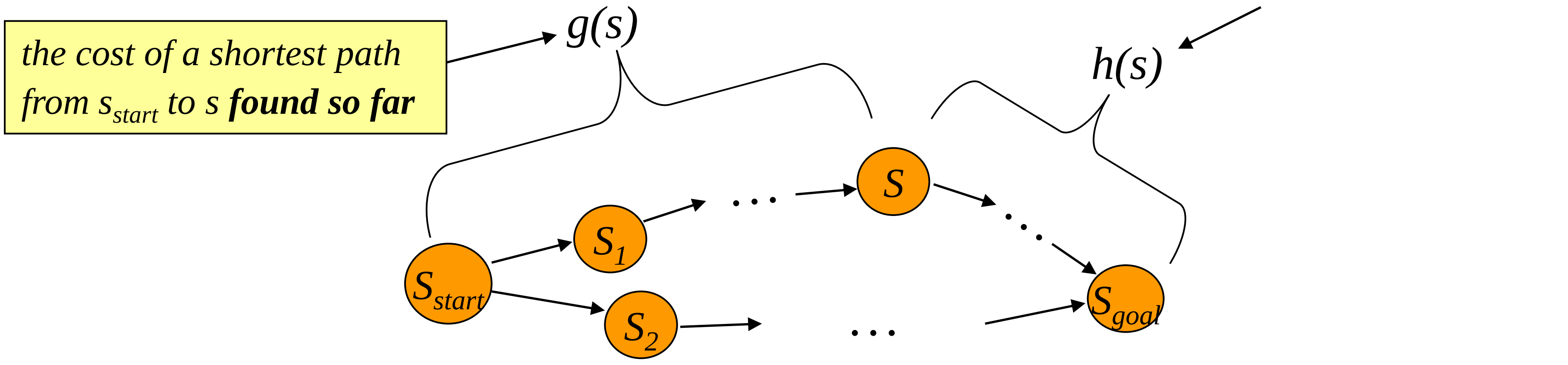
Effect of the Heuristic Function

- 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

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***

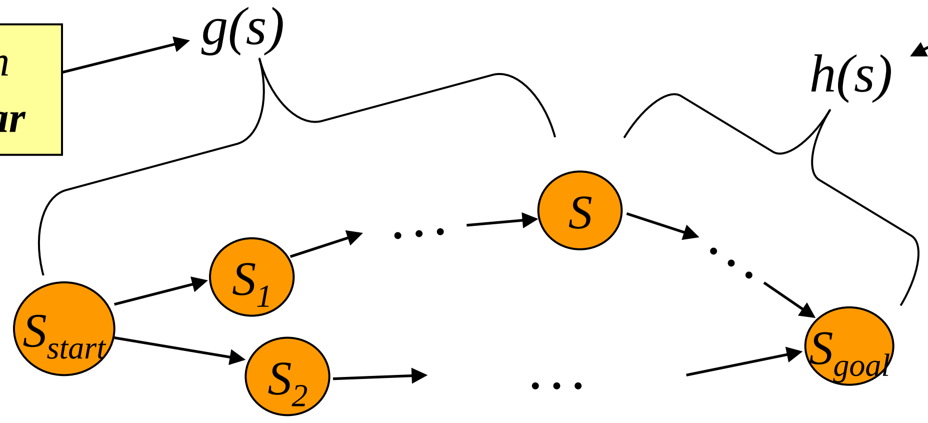


Effect of the Heuristic Function

- 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}

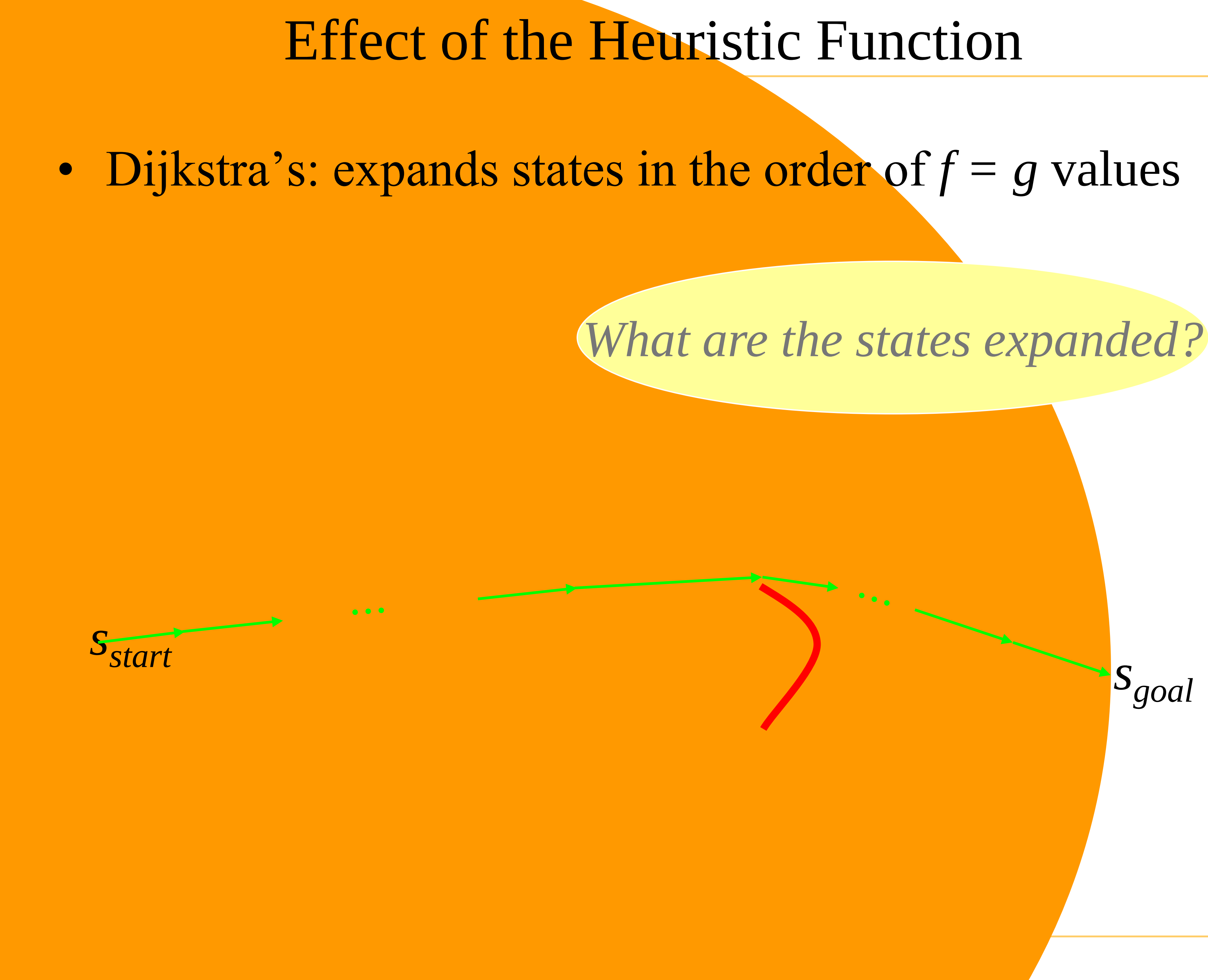
*the cost of a shortest path from s_{start} to s **found so far***



Effect of the Heuristic Function

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

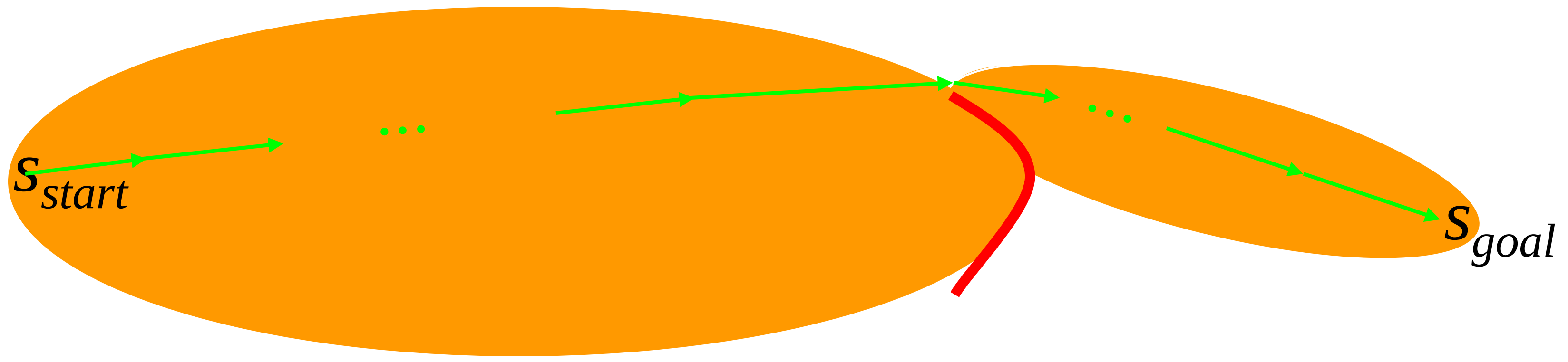
What are the states expanded?



Effect of the Heuristic Function

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

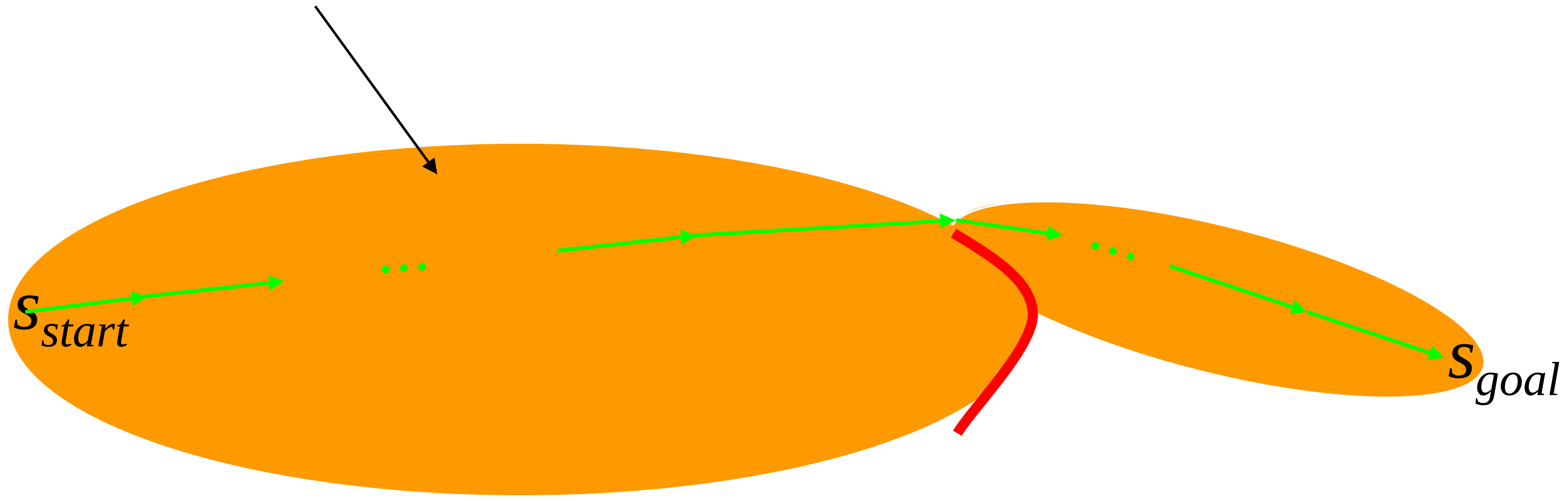
What are the states expanded?



Effect of the Heuristic Function

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

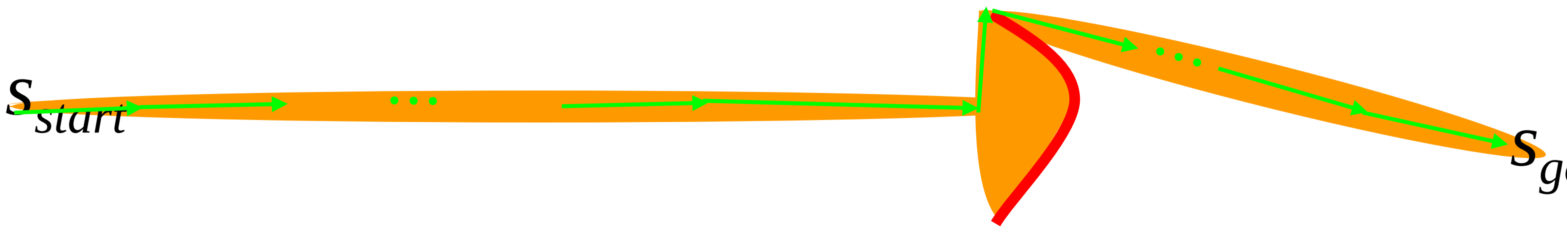
for large problems this results in A being slow*



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

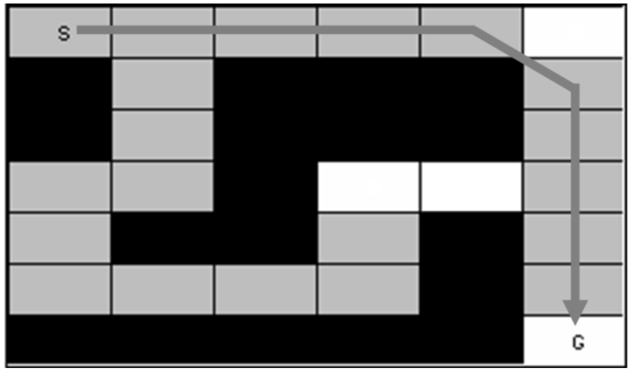


Effect of the Heuristic Function

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

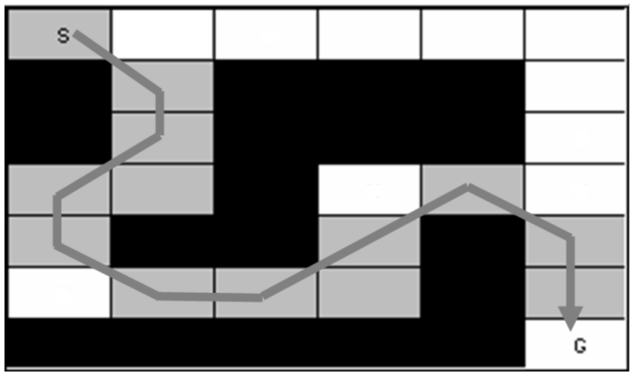
    $cost(solution) \leq \varepsilon \cdot cost(optimal\ solution)$
  - in many domains, it has been shown to be orders of magnitude faster than A*

A: ε =1.0*



20 expansions
solution=10 moves

Weighted A: ε =2.5*



13 expansions
solution=11 moves