15-466
Computer Game Programming

Basic Path Finding

Maxim Likhachev
Robotics Institute
Carnegie Mellon University
Path Planning

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

Why not pre-compute all paths?
Path Planning

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

\textit{re-planning path as map becomes known}
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 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
  - expand all obstacles by the radius of the object \( r \)
  - planning can be done for a point \( R \) (and not a circle anymore)

Is this a correct expansion?
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
- 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

• Skeletonization of the environment/C-Space
  - 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

1. Discretize the planning map
2. Convert the map into a graph
3. 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

\[ S_1 S_2 S_3 \]
\[ S_4 S_5 S_6 \]

*eight-connected grid (one way to construct a graph)*

convert into a graph

discretize

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

\[\text{discretize}\]
Planning via Cell Decomposition

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

convert into a graph

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

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)

\[
\begin{array}{ccc}
S_1 & S_2 & S_3 \\
S_4 & S_5 & S_6 \\
\end{array}
\]

so, what’s the solution?

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

convert into a graph
Planning via Cell Decomposition

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

\[ S_1 \quad S_2 \quad S_3 \]
\[ S_4 \quad S_5 \quad S_6 \]

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:
  - solution 2:
    - make the discretization adaptive
    - various ways possible

How?

convert into a graph

search the graph for a least-cost path from $s_{\text{start}}$ to $s_{\text{goal}}$
Planning via Cell Decomposition

- Graph construction:
  - connect neighbors

\[ S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad S_6 \]

*eight-connected grid*

convert into a graph
Planning via Cell Decomposition

• Graph construction:
  - connect neighbors
  - path is restricted to 45° degrees
Planning via Cell Decomposition

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

Will planning in 3D help?
Planning via Cell Decomposition

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

16-connected grid

convert into a graph
Planning via Cell Decomposition

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

Disadvantages?

16-connected grid

<table>
<thead>
<tr>
<th>S₁</th>
<th>S₂</th>
<th>S₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S₄</td>
<td>S₅</td>
</tr>
<tr>
<td>S₆</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 online
Planning via Cell Decomposition

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

**action template**

replicate it 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 \( <x, y, \text{orientation}, \text{velocity}> \)
  each edge represents a short feasible motion between corresponding cells

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

![Diagram of C-space or environment with suboptimal path and start and goal configurations](image-url)
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

\[ C\text{-}space \text{ or environment} \]

Assumption?

*suboptimal path*

*start configuration*

*goal configuration*
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

  \[ C\text{-space or environment} \]

  \textit{suboptimal path}

  \textit{start configuration}

  \textit{goal configuration}

\textbf{Assumption?}

\textbf{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 S. 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 S. 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 S. 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 S. Russell & 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 S. 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 S. 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. Funge
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?*

*from “Artificial Intelligence for Games” by I. Millington & J. Funge*
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.
A* Search

- Computes optimal g-values for relevant states at any point of time:

\[ g(s) \]

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

\[ h(s) \]

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

• Heuristic function must be:
  – admissible: for every state \( s \), \( h(s) \leq c^*(s,s_{goal}) \)
  – consistent (satisfy triangle inequality):
    \[
    h(s_{goal},s_{goal}) = 0 \text{ and for every } s \neq s_{goal}, h(s) \leq c(s,succ(s)) + h(succ(s))
    \]
  – admissibility follows from consistency and often consistency follows from admissibility

\[
\text{minimal cost from } s \text{ to } s_{goal}
\]
A* Search

- Computes optimal g-values for relevant states

**Main function**

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

ComputePath();
publish solution;

**ComputePath function**

while \( s_{\text{goal}} \) 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)

---

**Diagram:**

- **Nodes:** \( S_{\text{start}}, S_1, S_2, S_3, S_4, S_{\text{goal}} \)
- **Edges:**
  - \( S_{\text{start}} \) to \( S_1 \) with \( g=\infty, h=2 \)
  - \( S_1 \) to \( S_3 \) with \( g=\infty, h=1 \)
  - \( S_1 \) to \( S_{\text{goal}} \) with \( g=\infty, h=0 \)
  - \( S_{\text{start}} \) to \( S_2 \) with \( g=0, h=3 \)
  - \( S_2 \) to \( S_3 \) with \( g=\infty, h=2 \)
  - \( S_2 \) to \( S_{\text{goal}} \) with \( g=\infty, h=0 \)
  - \( S_4 \) to \( S_3 \) with \( g=\infty, h=1 \)

- **Costs:**
  - \( g(s) \) and \( h(s) \) values are shown at each node.
Computes optimal g-values for relevant states

**ComputePath function**

while ($s_{\text{goal}}$ is not expanded)

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

expand $s$;
• 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;

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

\[ g(s') = g(s) + c(s,s') \]

insert $s'$ into OPEN;

\[ \textit{CLOSED} = \{ \} \]

\[ \textit{OPEN} = \{ s_{start} \} \]

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: $s_1$
• Computes optimal g-values for relevant states

**ComputePath function**
while ($s_{\text{goal}}$ 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;

$\text{CLOSED} = \{s_{\text{start}}, s_2, s_1\}$
$\text{OPEN} = \{s_4, s_{\text{goal}}\}$
next state to expand: $s_4$
• Computes optimal g-values for relevant states

**ComputePath function**

\[
\text{while}(s_{\text{goal}} \text{ is not expanded})
\]

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 \(OPEN\);

\[
\text{CLOSED} = \{s_{\text{start}}, s_2, s_1, s_4\}
\]
\[
\text{OPEN} = \{s_3, s_{\text{goal}}\}
\]
next state to expand: \(s_{\text{goal}}\)
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 $g(s') > g(s) + c(s,s')$

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

insert $s'$ into OPEN;

---

expansion of $s$
Effect of the Heuristic Function

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

\[ g(s) \]
\[ h(s) \]
\[ an \ (under) \ estimate \ of \ the \ cost \ of \ a \ shortest \ path \ from \ s \ to \ s_{goal} \]

the cost of a shortest path from \( s_{start} \) to \( s \) found so far
Effect of the Heuristic Function

- 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

$S_{\text{start}}$  $S_{\text{goal}}$
Effect of the Heuristic Function

- **Weighted A* Search:**
  - trades off optimality for speed
  - $\varepsilon$-suboptimal: 
    \[
    \text{cost(solution)} \leq \varepsilon \cdot \text{cost(optimal solution)}
    \]
  - in many domains, it has been shown to be orders of magnitude faster than A*

\[
A^*: \varepsilon = 1.0 \\
20 \text{ expansions} \\
solution=10 \text{ moves}
\]

\[
\text{Weighted A*}: \varepsilon = 2.5 \\
13 \text{ expansions} \\
solution=11 \text{ moves}
\]