Search-based Planning with Motion Primitives
Maxim Likhachev
Carnegie Mellon University
What is Search-based Planning

• generate a graph representation of the planning problem
• search the graph for a solution
• can interleave the construction of the representation with the search (i.e., construct only what is necessary)

2D grid-based graph representation for 2D \((x,y)\) search-based planning:

lattice-based graph representation for 3D \((x,y,\theta)\) planning:

motion primitives
Search-based Planning Library (SBPL)

- http://www.ros.org/wiki/sbpl

- SBPL is:
  - a library of domain-independent graph searches
  - a library of environments (planning problems) that represent the problems as graph search problems
  - designed to be so that the same graph searches can be used to solve a variety of environments (graph searches and environments are independent of each other)
  - a standalone library that can be used with or without ROS and under linux or windows
Search-based Planning Library (SBPL)

- http://www.ros.org/wiki/sbpl

- SBPL can be used to:
  - implement particular planning modules such as $x,y,\theta$ planning and arm motion planning modules within ROS
  - design and drop-in new environments (planning problems) that represent the problem as a graph search and can therefore use existing graph searches to solve them
  - design and drop-in new graph searches and test their performance on existing environments

**Planning module**

- receives map, pose and goal updates
- updates environment (graph)
- calls graph search to re-plan

![Diagram of planning module](image)
Currently implemented graph searches within SBPL:
- ARA* - anytime version of A*
- Anytime D* - anytime incremental version of A*
- R* - a randomized version of A* (hybrid between deterministic searches and sampling-based planning)

Currently implemented environments (planning problems) within SBPL:
- 2D (x,y) grid-based planning problem
- 3D (x,y,θ) lattice-based planning problem
- 3D (x,y,θ) lattice-based planning problem with 3D (x,y,z) collision checking
- N-DOF planar robot arm planning problem

ROS packages that use SBPL:
- SBPL lattice global planner for (x,y,θ) planning for navigation
- SBPL cart planner for PR2 navigating with a cart
- SBPL motion planner for PR2 arm motions
- default move_base invokes SBPL lattice global planner as part of escape behavior

Unreleased ROS packages and other planning modules that use SBPL:
- SBPL door planning module for PR2 opening and moving through doors
- SBPL planning module for navigating in dynamic environments
- 4D planning module for aerial vehicles (x,y,z,θ)
...
What I will talk about

- Graph representations (implemented as environments for SBPL)
  - 3D \((x,y,\theta)\) lattice-based graph (within SBPL)
  - 3D \((x,y,\theta)\) lattice-based graph for 3D \((x,y,z)\) spaces (within SBPL)
  - Cart planning (separate SBPL-based package)
  - Lattice-based arm motion graph (separate SBPL-based motion planning module)
  - Door opening planning (separate SBPL-based package)

- Graph searches (implemented within SBPL)
  - ARA* - anytime version of A*
  - Anytime D* - anytime incremental version of A*
  - R* - a randomized version of A* (will not talk about)

- Heuristic functions (implemented as part of environments)

- Overview of how SBPL code is structured

- What’s coming
Lattice-based Graphs for Navigation

- Problems with (very popular) pure grid-based planning

2D grid-based graph representation for 2D (x,y) search-based planning:

\[ \text{discretize:} \quad \text{construct the graph:} \quad \text{search the graph for solution:} \]

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

*sharp turns do not incorporate the kinodynamics constraints of the robot*
Lattice-based Graphs for Navigation

- Problems with (very popular) pure grid-based planning

2D grid-based graph representation for 2D \((x,y)\) search-based planning:

3D-grid \((x,y,\theta)\) would help a bit but won’t resolve the issue
Lattice-based Graphs for Navigation

- Graphs constructed using motion primitives [Pivtoraiko & Kelly, ‘05]

outcome state is the center of the corresponding cell in the underlying \((x,y,\theta,...)\) cell

set of motion primitives pre-computed for each robot orientation (action template)

each transition is feasible (constructed beforehand)

replicate it online by translating it
Lattice-based Graphs for Navigation

- Graphs constructed using motion primitives [Pivtoraiko & Kelly, '05]
  - pros: sparse graph, feasible paths, can incorporate a variety of constraints
  - cons: possible incompleteness

set of motion primitives
pre-computed for each robot orientation
(action template)

replicate it
online
by translating it
Lattice-based Graphs for Navigation

- Graphs constructed using motion primitives [Pivtoraiko & Kelly, ‘05]
  - pros: sparse graph, feasible paths, can incorporate a variety of constraints
  - cons: possible incompleteness

planning on 4D ($<x,y,orientation,velocity>$) multi-resolution lattice using Anytime D*
[Likhachev & Ferguson, ‘09]

part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race
Lattice-based Graphs for Navigation

- Graphs constructed using motion primitives [Pivtoraiko & Kelly, ‘05]
  - pros: sparse graph, feasible paths, can incorporate a variety of constraints
  - cons: possible incompleteness

planning in 8D (foothold planning) lattice-based graph for quadrupeds [Vernaza et al., ’09]
using R* search [Likhachev & Stentz, ‘08]
Lattice-based Graphs for Navigation

- 3D \((x,y,\theta)\) lattice-based graph representation (*environment_navxythetalat.h/cpp in SBPL*)
  - takes set of motion primitives as input (.mprim files generated within matlab/mprim directory using corresponding matlab scripts):

  \(\text{unicycle model} \quad \text{or} \quad \text{unicycle with sideways motions} \quad \text{or} \quad \ldots\)

- takes the footprint of the robot defined as a polygon as input
Lattice-based Graphs for Navigation

- 3D \((x,y,\theta)\) lattice-based graph representation for 3D \((x,y,z)\) spaces
  (\textit{environment\_navxythetamlevlat.h/cpp in SBPL})
  - takes set of motion primitives as input
  - takes \(N\) footprints of the robot defined as polygons as input.
  - each footprint corresponds to the projection of a part of the body onto \(x,y\) plane.
  - collision checking/cost computation is done for each footprint at the corresponding projection of the 3D map
Graph Representation for Cart Planning

[Scholz, Marthi, Chitta & Likhachev, in submission]

- 3D \((x,y,\theta,\theta_{cart})\) lattice-based graph representation (in a separate Cart Planner package)
  - takes set of motion primitives feasible for the coupled robot-cart system as input (arm motions generated via IK)
  - takes footprints of the robot and the cart defined as polygons as input
Graph Representation for Arm Planning
[Cohen, Chitta & Likhachev, ICRA’10; Cohen et al., in submission]

- 7D (joint angles) lattice-based graph representation (in a separate SBPL Arm Planner package)
  - takes set of motion primitives defining joint angle changes as input
  - takes joint angle limits and link widths
  - goal is a 6 DoF pose for the end-effector
Graph Representation for Door Opening Planning

[Chitta, Cohen & Likhachev, ICRA’10]

• 4D \((x, y, \theta, \text{door interval})\) graph representation (in a separate SBPL Door Planner package)
  - takes set of motion primitives defining feasible \(x, y, \theta, \text{door angles}\) in the door frame as input
  - goal is for the door to be fully open
  - suitable for pushing/pulling doors
What I will talk about

- Graph representations (implemented as environments for SBPL)
  - 3D \((x, y, \theta)\) lattice-based graph (within SBPL)
  - 3D \((x, y, \theta)\) lattice-based graph for 3D \((x, y, z)\) spaces (within SBPL)
  - Cart planning (separate SBPL-based package)
  - Lattice-based arm motion graph (separate SBPL-based motion planning module)
  - Door opening planning (separate SBPL-based package)

- Graph searches (implemented within SBPL)
  - ARA* - anytime version of A*
  - Anytime D* - anytime incremental version of A*
  - R* - a randomized version of A* (will not talk about)

- Heuristic functions (implemented as part of environments)

- Overview of how SBPL code is structured

- What’s coming
Searching Graphs

• Once a graph is given (defined by environment file in SBPL), we need to search it for a path that minimizes cost as much as possible
• Many searches work by computing optimal $g$-values for relevant states

- $g(s)$ – an estimate of the cost of a least-cost path from $s_{\text{start}}$ to $s$

- Optimal values satisfy: $g(s) = \min_{s'' \in \text{pred}(s)} g(s'') + c(s'', s)$

![Diagram of searching graphs](image)
• 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: 
  \[
  g(s) = \min_{s'' \in \text{pred}(s)} g(s'') + c(s'', s)
  \]

the cost \( c(s_1, s_{goal}) \) of an edge from \( s_1 \) to \( s_{goal} \)
• 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
    
    $$
    s' = \arg \min_{s'' \in \text{pred}(s)} (g(s'') + c(s'', s))
    $$

![Diagram of searching graphs](image-url)
A* Search

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

\[ \text{the cost of a shortest path from } S_{\text{start}} \text{ to } s \text{ found so far} \]

\[ g(s) \]

\[ h(s) \]

an (under) estimate of the cost of a shortest path from \( s \) to \( s_{\text{goal}} \)
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):
    \[ h(s_{goal},s_{goal}) = 0 \text{ and for every } s \neq s_{goal}, h(s) \leq c(s,\text{succ}(s)) + h(\text{succ}(s)) \]
  – admissibility follows from consistency and often consistency follows from admissibility
A* Search

• Computes optimal g-values for relevant states

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

ComputePath function
while (\text{s}_{\text{goal}} \text{ is not expanded})
\hspace{1em} \text{remove } s \text{ with the smallest } [f(s) = g(s)+h(s)] \text{ from OPEN; }
\hspace{1em} \text{expand } s;

set of candidates for expansion

for every expanded state
\[ g(s) \text{ is optimal} \]
(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 Diagram]

- $S_{start}$
  - $g=0$
  - $h=3$

- $S_2$
  - $g=\infty$
  - $h=2$

- $S_1$
  - $g=\infty$
  - $h=1$

- $S_4$
  - $g=\infty$
  - $h=2$

- $S_3$
  - $g=\infty$
  - $h=1$

- $S_{goal}$
  - $g=\infty$
  - $h=0$
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;

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

\textbf{ComputePath function}

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

\begin{itemize}
  \item remove $s$ with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
  \item insert $s$ into $CLOSED$;
  \item for every successor $s'$ of $s$ such that $s'$ not in $CLOSED$
    \begin{itemize}
      \item if $g(s') > g(s) + c(s,s')$
        \begin{align*}
          g(s') &= g(s) + c(s,s');
          \text{insert } s' \text{ into } OPEN;
        \end{align*}
    \end{itemize}
\end{itemize}

$CLOSED = \{\}$

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

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;

\[ g=0 \]
\[ h=3 \]
\[ S_{start} \]

\[ g=1 \]
\[ h=2 \]
\[ S_2 \]

\[ g=\infty \]
\[ h=1 \]
\[ S_1 \]

\[ g=\infty \]
\[ h=0 \]
\[ S_{goal} \]

\[ g=\infty \]
\[ h=2 \]
\[ S_4 \]

\[ g=\infty \]
\[ h=1 \]
\[ S_3 \]
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($s_{\text{goal}}$ 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;

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

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

$\text{CLOSED} = \{s_{\text{start}}, s_2, s_1, s_4, s_{\text{goal}}\}$
$\text{OPEN} = \{s_3\}$
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 ($s_{\text{goal}}$ 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;

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($s_{\text{goal}}$ 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;

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

\[ g = 0 \quad h = 3 \]
\[ g = 2 \quad h = 2 \]
\[ g = 3 \quad h = 1 \]
\[ g = 5 \quad h = 0 \]
\[ g = 0 \quad h = 3 \]
\[ g = 2 \quad h = 2 \]
\[ g = 5 \quad h = 1 \]
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 $s_{\text{start}}$ to $s$ found so far

* an (under) estimate of the cost of a shortest path from $s$ to $s_{\text{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, $\varepsilon > 1 = \text{bias towards states that are closer to goal}$

The cost of a shortest path from $s_{\text{start}}$ to $s_{\text{found so far}}$ is $g(s)$.

An (under) estimate of the cost of a shortest path from $s$ to $s_{\text{goal}}$ is $h(s)$. 

\[ \begin{align*}
S_{\text{start}} & \quad \rightarrow \quad S_1 \quad \rightarrow \quad \ldots \\
S_1 & \quad \rightarrow \quad S \quad \rightarrow \quad \ldots \\
S_2 & \quad \rightarrow \quad \ldots \\
\ldots & \\
S_{\text{goal}} & \quad \rightarrow \quad \ldots
\end{align*} \]
Effect of the Heuristic Function

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

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

**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
  - $\varepsilon$-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*
  - 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
  – $\varepsilon$-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*
  – research becomes to develop a heuristic function that has shallow local minima
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 $\varepsilon$:

  $\varepsilon = 2.5$
  - 13 expansions
  - solution = 11 moves

  $\varepsilon = 1.5$
  - 15 expansions
  - solution = 11 moves

  $\varepsilon = 1.0$
  - 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 $\epsilon$:

$$\epsilon = 2.5$$  
13 expansions  
solution = 11 moves

$$\epsilon = 1.5$$  
15 expansions  
solution = 11 moves

$$\epsilon = 1.0$$  
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 $\varepsilon$:

  $\varepsilon = 2.5$
  13 expansions
  solution = 11 moves

  $\varepsilon = 1.5$
  15 expansions
  solution = 11 moves

  $\varepsilon = 1.0$
  20 expansions
  solution = 10 moves

• ARA* [Likhachev, Gordon & Thrun, ‘04]
  - an efficient version of the above that reuses state values within any search iteration
  - uses incremental version of A*
Other Motivation for Incremental A*

- Reuse state values from previous searches

\[ \text{cost of least-cost paths to } s_{\text{goal}} \text{ initially} \]

\[
\begin{array}{cccccccccccccccc}
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

\[ \text{cost of least-cost paths to } s_{\text{goal}} \text{ after the door turns out to be closed} \]

\[
\begin{array}{cccccccccccccccc}
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]
Other Motivation for Incremental A*

- Reuse state values from previous searches

\[
\text{cost of least-cost paths to } s_{\text{goal}} \text{ initially}
\]

\[
\begin{array}{cccccccccccc}
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 6 & 6 & 6 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
\end{array}
\]

\[
\text{cost of least-cost paths to } s_{\text{goal}} \text{ after the door turns out to be closed}
\]

\[
\begin{array}{cccccccccccc}
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 5 & 5 \\
\end{array}
\]

These costs are optimal g-values if search is done backwards.
Other Motivation for Incremental A*

- Reuse state values from previous searches

\[ \text{cost of least-cost paths to } s_{\text{goal}} \text{ initially} \]

<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

These costs are optimal g-values if search is done backwards.

Can we reuse these g-values from one search to another? – incremental A*

Cost of least-cost paths to \( s_{\text{goal}} \):

<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

These costs are optimal g-values if search is done backwards.

Can we reuse these g-values from one search to another? – incremental A*
Use of Incremental A* in D* Lite [Koenig & Likhachev, ‘02]

- Reuse state values from previous searches

*initial search by backwards A*

*initial search by D* Lite*

*second search by backwards A*

*second search by D* Lite*
A* with Reuse of State Values

• Alternative view of A*

all v-values initially are infinite;

ComputePath function
while(f(s_{goal}) > minimum f-value in OPEN )
  remove s with the smallest [g(s)+ h(s)] from OPEN;
  insert s into CLOSED;
  for every successor s’ of s
    if g(s’) > g(s) + c(s,s’)
      g(s’) = g(s) + c(s,s’);
      insert s’ into OPEN;

**A* with Reuse of State Values**

- Alternative view of A*

  all $v$-values initially are infinite;

  **ComputePath function**
  
  while ($f(s_{goal}) > \text{minimum } f\text{-value in OPEN}$)
  
  remove $s$ with the smallest $[g(s) + h(s)]$ from OPEN;
  
  insert $s$ into CLOSED;
  
  $v(s) = g(s);$  

  for every successor $s'$ of $s$
  
  if $g(s') > g(s) + c(s,s')$
    
    $g(s') = g(s) + c(s,s')$;
  
  insert $s'$ into OPEN;

\*v-value – the value of a state during its expansion (infinite if state was never expanded)
A* with Reuse of State Values

• Alternative view of A*

all \( v \)-values initially are infinite;

**ComputePath function**

while(\( f(s_{goal}) > \) minimum \( f \)-value in \( OPEN \))
  remove \( s \) with the smallest \( [g(s) + h(s)] \) from \( OPEN \);
  insert \( s \) into \( CLOSED \);
\( v(s) = g(s) \);

for every successor \( s' \) of \( s \)
  if \( g(s') > g(s) + c(s,s') \)
    \( g(s') = g(s) + c(s,s') \);
    insert \( s' \) into \( OPEN \);

• \( g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'',s') \)
A* with Reuse of State Values

• Alternative view of A*

all \( v \)-values initially are infinite;

**ComputePath function**

while \((f(s_{goal}) > \text{minimum } f\text{-value in } OPEN \) )

remove \( s \) with the smallest \([g(s) + h(s)] \) from \( OPEN \);

insert \( s \) into \( CLOSED \);

\( v(s) = g(s) \);

for every successor \( s' \) of \( s \)

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

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

insert \( s' \) into \( OPEN \);

- \( g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'',s') \)
A* with Reuse of State Values

• Alternative view of A*

all \( v \)-values initially are infinite;

**ComputePath function**

while(\( f(s_{goal}) > \) minimum \( f \)-value in OPEN )

remove \( s \) with the smallest \[ g(s) + h(s) \] from OPEN;

insert \( s \) into CLOSED;

\( v(s) = g(s) \);

for every successor \( s' \) of \( s \)

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

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

insert \( s' \) into OPEN;

\( g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'',s') \)

• OPEN: a set of states with \( v(s) > g(s) \)

all other states have \( v(s) = g(s) \)
A* with Reuse of State Values

• Alternative view of A*

all $v$-values initially are infinite;

ComputePath function

while($f(s_{goal}) > \text{minimum } f$-value in OPEN)

remove $s$ with the smallest $[g(s) + h(s)]$ from OPEN;
insert $s$ into CLOSED;
$v(s) = g(s)$;

for every successor $s'$ of $s$

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

$g(s') = g(s) + c(s,s')$;
insert $s'$ into OPEN;

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

• OPEN: a set of states with $v(s) > g(s)$
all other states have $v(s) = g(s)$
A* with Reuse of State Values

• Alternative view of A*

all \( v \)-values initially are infinite;

ComputePath function

while(\( f(s_{goal}) > \) minimum \( f \)-value in OPEN) 

remove \( s \) with the smallest \( [g(s)+ h(s)] \) from OPEN;

insert \( s \) into CLOSED;

\( v(s)=g(s) \);

for every successor \( s' \) of \( s \)

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

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

insert \( s' \) into OPEN;

\( g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'', s') \)

• OPEN: a set of states with \( v(s) > g(s) \)

all other states have \( v(s) = g(s) \)

• this A* expands overconsistent states in the order of their \( f \)-values
A* with Reuse of State Values

• Making A* reuse old values:

initialize $OPEN$ with all overconsistent states;

$ComputePathWithReuse$ function

while($f(s_{goal}) > \text{minimum } f\text{-value in } OPEN$ )
    remove $s$ with the smallest $[g(s) + h(s)]$ from $OPEN$;
    insert $s$ into $CLOSED$;

$v(s)=g(s)$;

for every successor $s'$ of $s$
    if $g(s') > g(s) + c(s,s')$
        $g(s') = g(s) + c(s,s')$;
        insert $s'$ into $OPEN$;

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

• $OPEN$: a set of states with $v(s) > g(s)$
    all other states have $v(s) = g(s)$

• this A* expands overconsistent states in the order of their $f$-values

all you need to do to make it reuse old values!
A* with Reuse of State Values

\[ g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'', s') \]

initially OPEN contains all overconsistent states

\[ \text{CLOSED} = {} \]
\[ \text{OPEN} = \{s_4, s_{\text{goal}}\} \]

next state to expand: \( s_4 \)
A* with Reuse of State Values

CLOSED = \{s_4\}
OPEN = \{s_3, s_{goal}\}
next state to expand: s_{goal}
**A* with Reuse of State Values**

**CLOSED** = \{s_4, s_{goal}\}

**OPEN** = \{s_3\}

**done**

*after ComputePathwithReuse terminates:*

all g-values of states are equal to final A* g-values
A* with Reuse of State Values

we can now compute a least-cost path
A* with Reuse of State Values

• Making weighted A* reuse old values:

initialize $OPEN$ with all overconsistent states;

**ComputePathwithReuse function**

while($f(s_{\text{goal}}) > \text{minimum } f\text{-value in } OPEN$)

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

insert $s$ into $CLOSED$;

$v(s) = g(s)$;

for every successor $s'$ of $s$

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

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

if $s'$ not in $CLOSED$ then insert $s'$ into $OPEN$;

just make sure no state is expanded multiple times
Anytime Repairing A* (ARA*)

- Efficient series of weighted A* searches with decreasing $\varepsilon$:
  
  set $\varepsilon$ to large value;
  
  $g(s_{start}) = 0$; $v$-values of all states are set to infinity; $OPEN = \{s_{start}\}$;

  while $\varepsilon \geq 1$

  \[
  CLOSED = \{\};
  \]
  ComputePathwithReuse();
  
  publish current $\varepsilon$ suboptimal solution;
  
  decrease $\varepsilon$;

  initialize $OPEN$ with all overconsistent states;
ARA*

Efficient series of weighted A* searches with decreasing $\varepsilon$:

set $\varepsilon$ to large value;
$g(s_{\text{start}}) = 0$; $v$-values of all states are set to infinity; $OPEN = \{s_{\text{start}}\}$;
while $\varepsilon \geq 1$

$CLOSED = \{\}$;
ComputePathwithReuse();
publish current $\varepsilon$ suboptimal solution;
decrease $\varepsilon$;
initialize $OPEN$ with all overconsistent states;

need to keep track of those
ARA*

- Efficient series of weighted A* searches with decreasing $\varepsilon$:

  initialize $OPEN$ with all overconsistent states;

  **ComputePathwithReuse** function

  while($f(s_{goal}) >$ minimum $f$-value in $OPEN$ )
  
  remove $s$ with the smallest $[g(s)+ \varepsilon h(s)]$ from $OPEN$;
  
  insert $s$ into $CLOSED$;
  
  $v(s)=g(s)$;

  for every successor $s'$ of $s$
  
  if $g(s') > g(s) + c(s,s')$
    
    $g(s') = g(s) + c(s,s')$;
    
    if $s'$ not in $CLOSED$ then insert $s'$ into $OPEN$;
    
    otherwise insert $s'$ into $INCONS$

  • $OPEN \cup INCONS = \text{ all overconsistent states}$
ARA*

- Efficient series of weighted A* searches with decreasing $\varepsilon$:
  set $\varepsilon$ to large value;
  $g(s_{\text{start}}) = 0; \ \nu$-values of all states are set to infinity; $OPEN = \{s_{\text{start}}\}$;
  while $\varepsilon \geq 1$
    $CLOSED = \{\}; \ \text{INCONS} = \{\}$;
    ComputePathwithReuse();
    publish current $\varepsilon$ suboptimal solution;
    decrease $\varepsilon$;
    initialize $OPEN = OPEN U \text{INCONS}$;

all overconsistent states (exactly what we need!)
ARA*

• A series of weighted A* searches

\[ \epsilon = 2.5 \]
13 expansions
solution = 11 moves

\[ \epsilon = 1.5 \]
15 expansions
solution = 11 moves

\[ \epsilon = 1.0 \]
20 expansions
solution = 10 moves

• ARA*

\[ \epsilon = 2.5 \]
13 expansions
solution = 11 moves

\[ \epsilon = 1.5 \]
1 expansion
solution = 11 moves

\[ \epsilon = 1.0 \]
9 expansions
solution = 10 moves
What I will talk about

• Graph representations (implemented as environments for SBPL)
  - 3D \((x,y,\theta)\) lattice-based graph (within SBPL)
  - 3D \((x,y,\theta)\) lattice-based graph for 3D \((x,y,z)\) spaces (within SBPL)
  - Cart planning (separate SBPL-based package)
  - Lattice-based arm motion graph (separate SBPL-based motion planning module)
  - Door opening planning (separate SBPL-based package)

• Graph searches (implemented within SBPL)
  - ARA* - anytime version of A*
  - Anytime D* - anytime incremental version of A*
  - R* - a randomized version of A* (will not talk about)

• Heuristic functions (implemented as part of environments)

• Overview of how SBPL code is structured

• What’s coming
Anytime and Incremental Planning

• **Anytime D**[^2008] [Likhachev et al., ‘2008]:
  - decrease $\varepsilon$ and update edge costs at the same time
  - re-compute a path by reusing previous state-values

set $\varepsilon$ to large value;
until goal is reached
  ComputePathwithReuse();  //modified to handle cost increases
  publish $\varepsilon$-suboptimal path;
  follow the path until map is updated with new sensor information;
  update the corresponding edge costs;
  set $s_{\text{start}}$ to the current state of the agent;
  if significant changes were observed
    increase $\varepsilon$ or replan from scratch;
  else
    decrease $\varepsilon$;
Anytime and Incremental Planning

• Anytime D* in Urban Challenge

planning on 4D (x,y,orientation,velocity) multi-resolution lattice using Anytime D*
[Likhachev & Ferguson, ‘09]

part of efforts by Tartanracing team from CMU for the Urban Challenge 2007 race
Other Uses of Incremental A*

• Whenever planning is a repeated process:
  – improving a solution (e.g., in anytime planning)
  – re-planning in dynamic and previously unknown environments
  – adaptive discretization
  – many other planning problems can be solved via iterative planning
What I will talk about

- Graph representations (implemented as environments for SBPL)
  - 3D (x,y,θ) lattice-based graph (within SBPL)
  - 3D (x,y,θ) lattice-based graph for 3D (x,y,z) spaces (within SBPL)
  - Cart planning (separate SBPL-based package)
  - Lattice-based arm motion graph (separate SBPL-based motion planning module)
  - Door opening planning (separate SBPL-based package)

- Graph searches (implemented within SBPL)
  - ARA* - anytime version of A*
  - Anytime D* - anytime incremental version of A*
  - R* - a randomized version of A* (will not talk about)

- Heuristic functions (implemented as part of environments)

- Overview of how SBPL code is structured

- What’s coming
Heuristic Functions

- 2D (x,y) Dijkstra’s taking into account all obstacles for:
  - 3D (x,y,θ) lattice-based graph
  - 3D (x,y,θ) lattice-based graph for 3D (x,y,z) spaces
  - cart planning

- Angle distance to the fully open door for:
  - door opening planning

- 3D (x,y,z) Dijkstra’s for the end-effector taking into account all obstacles for:
  - lattice-based arm motion graph (separate SBPL-based motion planning module)
What I will talk about

- Graph representations (implemented as environments for SBPL)
  - 3D \((x,y,\theta)\) lattice-based graph (within SBPL)
  - 3D \((x,y,\theta)\) lattice-based graph for 3D \((x,y,z)\) spaces (within SBPL)
  - Cart planning (separate SBPL-based package)
  - Lattice-based arm motion graph (separate SBPL-based motion planning module)
  - Door opening planning (separate SBPL-based package)

- Graph searches (implemented within SBPL)
  - ARA* - anytime version of A*
  - Anytime D* - anytime incremental version of A*
  - R* - a randomized version of A* (will not talk about)

- Heuristic functions (implemented as part of environments)

- Overview of how SBPL code is structured

- What’s coming
Structure of SBPL

- Example configuration files to run main.cpp on. These can be used to test planners outside of ROS, just running compiled test program in bin directory from the command line.

- Motion primitives used as input for 3D \((x,y,\theta)\) lattice-based planning and matlab files to generate new motion primitives.

- Few matlab scripts to visualize 3D paths and motion primitives.

- Environments (.cpp and .h files defining planning problems as graphs).

- Domain-independent graph searches.

- For compiling under Linux/Windows using CMake.

- For compiling under Windows using Visual Studio.
Structure of SBPL

Environment represented as a graph (<x,y,θ> planning, arm planning, etc.)
graph constructed on the fly

ID’s of start and goal states
ID’s of successor states, transition costs,…
heuristics

Graph search
(ARA*, Anytime D*, etc.)
memory allocated dynamically

request for ID’s of successors states and transition costs during graph search
requests for heuristics
plan as a sequence of state ID’s
Structure of SBPL

• Look at Main.cpp for examples for how to use SBPL:

```cpp
EnvironmentNAVXYTHETALAT environment_navxythetalat;
if(!environment_navxythetalat.InitializeEnv(argv[1], perimeterptsV, NULL))
{
    SBPL_ERROR("ERROR: InitializeEnv failed\n");
    throw new SBPL_Exception();
}
if(!environment_navxythetalat.InitializeMDPCfg(&MDPCfg))
{
    SBPL_ERROR("ERROR: InitializeMDPCfg failed\n");
    throw new SBPL_Exception();
}
//plan a path
vector<int> solution_stateIDs_V;
bool bforwardsearch = false;
ADPlanner planner(&environment_navxythetalat, bforwardsearch);
if(planner.set_start(MDPCfg.startstateid) == 0)
{
    SBPL_ERROR("ERROR: failed to set start state\n");
    throw new SBPL_Exception();
}
if(planner.set_goal(MDPCfg.goalstateid) == 0)
{
    SBPL_ERROR("ERROR: failed to set goal state\n");
    throw new SBPL_Exception();
}
planner.set_initialsolution_eps(3.0);

bRet = planner.replan(allocated_time_secs, &solution_stateIDs_V);
SBPL_PRINTF("size of solution=%d\n", (unsigned int)solution_stateIDs_V.size());
```
What I will talk about

- Graph representations (implemented as environments for SBPL)
  - 3D (x,y,θ) lattice-based graph (within SBPL)
  - 3D (x,y,θ) lattice-based graph for 3D (x,y,z) spaces (within SBPL)
  - Cart planning (separate SBPL-based package)
  - Lattice-based arm motion graph (separate SBPL-based motion planning module)
  - Door opening planning (separate SBPL-based package)

- Graph searches (implemented within SBPL)
  - ARA* - anytime version of A*
  - Anytime D* - anytime incremental version of A*
  - R* - a randomized version of A* (will not talk about)

- Heuristic functions (implemented as part of environments)

- Overview of how SBPL code is structured

- What’s coming
What’s coming

- Planning in Dynamic Environments
- Planning for Spring-loaded Doors
- ROS package for \((x,y,\theta)\) planning while accounting for the whole body of PR2 in 3D \((x,y,z)\)
http://www.ros.org/wiki/sbpl

Thanks to Willow Garage for the support of SBPL!