Canonical Orderings on Grids

Nathan R. Sturtevant
University of Denver

Steve Rabin
DigiPen Institute of Technology
A*  Jump Point Search
A*  

Jump Point Search
Contributions

• Jump Point Search (JPS)
  • Harabor and Grastien: 2011; 2014
Contributions

• Jump Point Search (JPS)
  • Harabor and Grastien: 2011; 2014
• Decompose JPS:
  • Canonical ordering of states
  • Jumping policy
  • Best first search
Contributions

• Jump Point Search (JPS)
  • Harabor and Grastien: 2011; 2014
• Decompose JPS:
  • Canonical ordering of states
  • Jumping policy
  • Best first search
• Construct new algorithms
  • Canonical A*, Canonical Dijkstra
  • Bounded JPS, Weighted JPS
Canonical ordering of paths

• Order all **optimal** paths:
  • Path $p_1$ is preferred over path $p_2$ if
    • $p_1$ has diagonal actions prior to $p_2$
Canonical ordering of paths

• Order all **optimal** paths:
  • Path $p_1$ is preferred over path $p_2$ if
    • $p_1$ has diagonal actions prior to $p_2$
Canonical ordering of paths

- Order all **optimal** paths:
  - Path $p_1$ is preferred over path $p_2$ if
    - $p_1$ has diagonal actions prior to $p_2$
Algorithm #1: Canonical A*

- Run regular A* using the canonical ordering
- Looks the same as A* on the full graph
- Slightly different node expansions
  - Tie-breaking at the goal
- Far fewer generations
## Experimental Results

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>A*</th>
<th>CA*</th>
<th>JPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.600</td>
<td>2.566</td>
<td>1.982</td>
</tr>
</tbody>
</table>

Apply in state spaces where generations are expensive!
Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>A*</th>
<th>CA*</th>
<th>JPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>5.600</td>
<td>2.566</td>
<td>1.982</td>
</tr>
<tr>
<td>Expansions</td>
<td>13,295</td>
<td>13,302</td>
<td>229</td>
</tr>
</tbody>
</table>

Apply in state spaces where generations are expensive!
# Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>A*</th>
<th>CA*</th>
<th>JPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>5.600</td>
<td>2.566</td>
<td>1.982</td>
</tr>
<tr>
<td>Expansions</td>
<td>13,295</td>
<td>13,302</td>
<td>229</td>
</tr>
<tr>
<td>Generations</td>
<td>99,483</td>
<td>13,654</td>
<td>61,282</td>
</tr>
</tbody>
</table>

Apply in state spaces where generations are expensive!
JPS Generations

A*/CA*

JPS
JPS Generations

A*/CA*

JPS
JPS Generations

A*/CA*

JPS
Jumping Policy

- Continually generate successors until you reach:
  - A jump point (open)
  - The goal (open)
  - A wall (discard)
Jumping Policy

• Continually generate successors until you reach:
  • A jump point (open)
  • The goal (open)
  • A wall (discard)
Algorithm #2: Canonical Dijkstra

• For fast Single-Source Shortest Path Computation
  • Use Canonical Ordering with Dijkstra
  • When we jump over states, write their g-cost to closed
## Experimental Results

<table>
<thead>
<tr>
<th>Map</th>
<th>Dijkstra</th>
<th>CA*</th>
<th>Canonical Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon Age</td>
<td>1.0</td>
<td>2.1</td>
<td>3.4</td>
</tr>
</tbody>
</table>

**Speedup Factor**
## Experimental Results

<table>
<thead>
<tr>
<th>Map</th>
<th>Dijkstra</th>
<th>CA*</th>
<th>Canonical Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon Age</td>
<td>1.0</td>
<td>2.1</td>
<td>3.4</td>
</tr>
<tr>
<td>Random</td>
<td>1.0</td>
<td>1.7</td>
<td>2.5</td>
</tr>
</tbody>
</table>
## Experimental Results

<table>
<thead>
<tr>
<th>Map</th>
<th>Dijkstra</th>
<th>CA*</th>
<th>Canonical Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon Age</td>
<td>1.0</td>
<td>2.1</td>
<td>3.4</td>
</tr>
<tr>
<td>Random</td>
<td>1.0</td>
<td>1.7</td>
<td>2.5</td>
</tr>
<tr>
<td>Starcraft</td>
<td>1.0</td>
<td>2.2</td>
<td>4.0</td>
</tr>
</tbody>
</table>

*Speedup Factor*
Algorithm #3: Bounded JPS

- Limit the length of jumps
- Parameterization between Canonical A* and JPS
  - No jumping is Canonical A*
  - Infinite jumping is JPS
# Experimental Results

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>CA*</th>
<th>BJPS(4)</th>
<th>JPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.59</td>
<td>1.50</td>
<td>1.98</td>
</tr>
</tbody>
</table>
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>CA*</th>
<th>BJPS(4)</th>
<th>JPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (ms)</strong></td>
<td>2.59</td>
<td>1.50</td>
<td>1.98</td>
</tr>
<tr>
<td><strong>Expansions</strong></td>
<td>13,301</td>
<td>4,091</td>
<td>229</td>
</tr>
</tbody>
</table>
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>CA*</th>
<th>BJPS(4)</th>
<th>JPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time (ms)</strong></td>
<td>2.59</td>
<td>1.50</td>
<td>1.98</td>
</tr>
<tr>
<td><strong>Expansions</strong></td>
<td>13,301</td>
<td>4,091</td>
<td>229</td>
</tr>
<tr>
<td><strong>Generations</strong></td>
<td>13,654</td>
<td>22,363</td>
<td>61,281</td>
</tr>
</tbody>
</table>
Best first search

- We can use suboptimal algorithms (e.g., Weighted A*) to search with JPS
- JPS only puts onto OPEN:
  - Jump points
  - The goal
Conclusions

• By decomposing JPS we:
  • Gain a better understanding of JPS
  • Are able to introduce new algorithms using the ideas of JPS
    • Bounded JPS
    • Canonical A*, Canonical Dijkstra
    • Weighted JPS

• http://www.movingai.com/