MULTI-AGENT NAVIGATION (PART 1)
In design, you can never choose whether you pay a cost, only how you pay it.

Dr. Fred Brooks
SEAN’S LECTURES

- Oct 2 – Introduction and Graph Searches
- Oct 7 – Intro to Menge, Git, and OpenGL
  - Assign homework 2
- Oct 9 – Global planners
- Oct 14 – Local Planners (part 1)
- Oct 21 – Local Planners (part 2)
  - Assign homework 3
- Oct 23 – Extraneous issues
ROADMAP SURVEY

• Languages
  • C/C++, Python, Java
• Representation
  • Objects/Pointers, Adjacency List, Matrix
• Even starting ground for next homework assignment
MULTI-AGENT NAVIGATION

• Why do it?
  • Autonomous cars
  • Robot assembly lines
  • Swarm simulation
  • Pedestrian simulation
MULTI-AGENT NAVIGATION

• Planning for multiple robots
  • *Can* be the same as for a single robot with multiple parts
    • The parts need not be connected
  • Dimension grows linearly with the robots
    • For N simple 2D, translational robots, there are 2N dimensions in configuration space
    • Algorithmic complexity tends to be exponential in dimensions (for “complete” solutions)
MULTI-AGENT NAVIGATION

• How do we do it?
  • Complete solutions are infeasible
  • “Decoupled” solutions
    • Independent solutions whose interactions are coordinated
  • Computational necessity
  • Design decision
    • Entities are often independent
MULTI-AGENT NAVIGATION

• Skipping general multi-agent navigation
  • Path coordination
  • Pareto optimality
  • Prioritized planning
  • We’ll come back to it
• Focus on pedestrian/crowd simulation
Simulation State: obstacles (static & dynamic), agents
Goal Selection: High-order model of what the agent wants
Static Planning: Plan to reach goal vs. static obstacles
Local Collision Avoidance: Adapt plan because of other agents
• We’ll have two homework assignments
  • Implement static planning algorithm
  • Implement local collision avoidance
STATIC PLANNING

- Identifying and encoding traversable space
  - Roadmaps
  - Navigation Mesh
  - Corridor Maps
  - Guidance/potential fields
- (We’ll talk about these in detail in a week).
STATIC PLANNING

• Graph searches
  • Many of the most common structures are, ultimately, graphs
  • Finding paths from start to end become a basic operation
  • Let’s look at path computation
    • http://www.youtube.com/watch?v=czk4xgdhdY4
    • http://www.youtube.com/watch?v=nDyGEq_uGGo
Typically, we’re looking not for any path. We have a sense of “optimality” and want to find the optimal path.

- Typically distance

Can be other functions: e.g.,

- Energy consumed (such as for uneven terrain)
- Psychological comfort (avoiding “negative” regions)
OPTIMAL PATH

• The roadmap (and all graph-based traversal structures) encode the costs of moving from one node to another.
  • Cost of movement is the edge weight.
• Given graph and optimality definition, how do we compute the optimal path?
OPTIMAL PATH

• Assumptions
  • The edge weights are non-negative
    • i.e., every section of the path requires a “cost”
    • No path section provides a “gain”
BREADTH/DEPTH-FIRST SEARCHES

• Depth-first
  • Similar to wall-following algorithms
• Breadth-first
  • Weights are ignored, the boundary of the search space is all nodes $k$ steps away from the source.
• This is guaranteed to find a path if one exists
• Only guaranteed to be optimal if it is the only path
DJIKSTRA’S ALGORITHM

- Single-source shortest-path (to all other nodes)
  - Shortest path to a specific target node is simply an early termination
  - Dijkstra’s Algorithm requires our non-negative cost assumption
  - What is the algorithm?

DJIKSTRA’S ALGORITHM

\[ \text{minDistance}( \text{start}, \text{end}, \text{nodes} ) \]

For all nodes \( n_i \), \( i \neq \text{start} \), \( \text{cost}(n_i) = \infty \)
\( \text{cost}(\text{start}) = 0 \)

\( \text{unvisited} = \text{nodes} \setminus \{\text{start}\} \) // set

\( c = \text{start} \) // current node

while ( true )
    if ( \( c == \text{end} \) ) return \( \text{cost}(c) \)
    For each unvisited neighbor, \( n \), of \( c \)
        \( \text{cost}(n) = \min( \text{cost}(n), \text{cost}(c) + E(c,n) ) \)
    \( c = \min\text{Cost}( \text{unvisited} ) \) // 1
    if ( \( \text{cost}(c) == \infty \) ) return \( \infty \)

Why?

1) We’ll say that \( \text{minCost} \) returns \( \infty \) if there are no nodes in the set.
DJIKSTRA’S ALGORITHM

• How do we modify it to get a path?
• What is the cost of this algorithm?
DJIKSTRA’S ALGORITHM

shortestPath( start, end, nodes )
  For all nodes n_i, i ≠ start
    cost(n_i) = ∞
    prev(n_i) = Ø
  cost( start ) = 0
  unvisited = nodes \ {start}  # set difference
  visited = {}
  c = start # current node
  while ( true )
    if ( c == end ) break
    For each unvisited neighbor, n, of c
      if ( cost(n) > cost(c) + E(c,n) )
        cost(n) = cost(c) + E(c,n)
        prev(n) = c
    c = minCost( unvisited )
    if ( cost( c ) == ∞ ) break
    if ( cost(end) < ∞ )
      construct path
DJIKSTRA’S ALGORITHM

• Constructing a path

```python
path = [ end ]
p = prev[ end ]
while (p != Ø)
    path = [ p ] + path  // list concatenation
    p = prev[ p ]
return path
```
DJIKSTRA’S ALGORITHM

• What is the cost of this algorithm?
• If the graph has $V$ vertices and $E$ edges:
  • $E \times d + V \times m$
    • $d$ is the cost to change a node’s cost
    • $m$ is the cost to extract the *minimum* unvisited node
  • $d$ is typically a nominal constant
  • $m$ depends on how we find the minimum node
DJIKSTRA’S ALGORITHM

• Minimum neighbor
  • Djikstra originally did a search through a list
    • Maintaining a sorted vector doesn’t solve the problem
    • The cost of maintaining the sort would be the same as simply searching
  • Cost was $|E| + |V|^2$
DJIKSTRA’S ALGORITHM

• Minimum neighbor
  • Use a good min-heap implementation and it becomes
    • $|E| + |V| \log |V|$
    • (Good $\rightarrow$ Fibonacci heap)

DJIKSTRA’S ALGORITHM

• Good general solution
  • Guaranteed to find optimal solution
  • Not very smart
  • Why?
DJIKSTRA’S ALGORITHM

• Djikstra’s algorithm expands the front uniformly
  • It extends the nearest node on the front
  • This causes the search space to inflate uniformly
A* ALGORITHM

• “Best-first” graph search algorithm
  • Uses a knowledgeable heuristic to estimate the cost of a node
  • At any given time, the *expected* cost of a node, \( f(x) \), is the sum of two terms
    • Its known cost from the start, \( g(x) \)
    • Its estimated cost to the goal, \( h(x) \)

A* ALGORITHM

• Admissible heuristics
  • \( h(x) \leq D(x,\text{goal}) \)
    • \( D(x,y) \) actual distance from node \( x \) to \( y \)
    • i.e., it must be a conservative estimate
  • In path planning, our heuristic is usually Euclidian distance
    • Triangle-inequality insures admissibility
  • \( h(x) \leq E(x,y) + h(y) \)
A* ALGORITHM

- Admissible heuristics
  - Monotonic/consistent
    - $h(x) \leq E(x,y) + h(y)$
    - i.e., the “best guess” for a node cannot be beaten by the known cost to move to another node and my best guess from there
  - This applies to our Euclidian distance heuristic
**A* ALGORITHM**

```
minDistance( start, end, nodes )
    closed = {}
    open = {start}
    g[ start ] = 0
    f[ start ] = g[ start ] + h( start, end )
while ( ! open.isEmpty() )
    c = minF( open )
    if ( c == end ) return g[ c ]
    open = open \ {c}; closed = closed U {c}
    for each neighbor, n, of c
        gTest = g[ c ] + E( n, c )
        fTest = gTest = h( n, e )
        if ( n in closed && fTest ≥ f[ n ] ) continue
        if ( n not in open || fTest < f[n] )
            g[ n ] = gTest
            f[ n ] = fTest
            open = open U {n}
```
A* ALGORITHM

- Closed set
  - It is (apparently) possible to visit a node but then later need to place it back in the open set.
    - \( f(n) = g(n) + h(n, e) \)
    - \( h(n, e) \) is constant for constant \( n \) & \( e \)
    - So, to revisit \( n \), \( f'(n) < f(n) \Rightarrow g'(n) < g(n) \)
    - We found a SHORTER path to that node
  - I cannot come up with a circumstance where this happens with the distance heuristic*

* Absence of proof is not proof of absence.
A* ALGORITHM

\[ \text{minDistance}( \text{start}, \text{end}, \text{nodes} ) \]

\[ \text{closed} = {} \]
\[ \text{open} = \{ \text{start} \} \]
\[ g[ \text{start} ] = 0 \]
\[ f[ \text{start} ] = g[ \text{start} ] + h( \text{start}, \text{end} ) \]

while ( !open.isEmpty() )

\[ c = \text{minF}( \text{open} ) \]
if ( \( c == \text{end} \) ) return \( g[ c ] \)

open = open \ \{ c \}; closed = closed \ U \ \{ c \}

for each neighbor, \( n \), of \( c \)

\[ \text{if} ( \ n \ \text{in} \ \text{closed} ) \ \text{continue} \]
\[ g\text{Test} = g[ c ] + E( n, c ) \]
if ( \( g\text{Test} < g[ n ] \) )

\[ g[ n ] = g\text{Test}; \ f[ n ] = g\text{Test} + h(n, \text{end}) \]

open = open \ U \ \{ n \}
A* ALGORITHM

• Notes

  • The goal node may be visited/updated multiple times
    • There may be multiple paths to it
  • Only when the goal node is the “closest” node is it considered final
  • Like Djikstra’s, it will still fall victim to local minima
    • But gets around them more efficiently
A* ALGORITHM

• Constructing a path
  • We add the same instrumentation
    • Record where we came from when we reduce the cost of each node
    • Construct the path by tracing backwards from the goal
A* ALGORITHM

• Efficient solution
  • Guaranteed to find optimal solution (for admissible heuristic)
  • Much more optimized search space
    • Can be fooled by adversarial graph
A* ALGORITHM

• Demos
  • http://www.youtube.com/watch?v=DINCL5cd_w0
WEIGHTED A* ALGORITHM

- $f(n) = g(n) + \varepsilon h(n)$
- $\varepsilon = 0 \rightarrow$ Djikstra’s algorithm
WEIGHTED A* ALGORITHM

• $f(n) = g(n) + \varepsilon h(n)$
• $\varepsilon = 1 \rightarrow A^*$ algorithm
WEIGHTED A* ALGORITHM

- \( f(n) = g(n) + \varepsilon h(n) \)
  - \( \varepsilon > 1 \) → Strong bias straight towards goal
  - Trades optimality for speed
    - Cost of path \( \leq \varepsilon \times \text{cost of optimal} \)
D* ALGORITHM

• These algorithms assume perfect \textit{a priori} knowledge of the environment.

• What if our knowledge of the environment (or the environment itself) changes over time?

• We use an incremental search algorithm

• D*, D*lite, etc.

• These algorithms used in the Mars rovers and the DARPA grand challenge winners

QUESTIONS?