ROMS Error Reduction Project

By Benjamin Davini

CSC 590 – Graduate Seminar
Advisor: Dr. Chris Clark
Help from: Dr. Paul Choboter
Introduction

- ROMS – Regional Ocean Modeling System
  - mathematical model capable of simulating currents, ecosystems, biogeochemical cycles, and sediment movement in various coastal regions
  - Very large system with a great number of equations and variables
Goal

- Reduce most efficiently the error in ROMS by utilizing AUVs to obtain measurements at the most optimal places
Concerning ourselves with a specific part of ROMS that seeks to reduce the value of a cost function, $J$, which represents the amount of error in a given simulation.
Introduction - Cost function

\[ J(x) = (x - x_{obs})^T \cdot O^{-1} \cdot (x - x_{obs}) + (x - x_b)^T \cdot B^{-1} \cdot (x - x_b) \]

Where:

- \( X_{obs} \): vector of observations
- \( O \): observation covariance matrix
- \( X_b \): predictions of the model without any data assimilation (background)
- \( B \): background error covariance matrix
Introduction - Cost function

... and $x$ is a vector containing the following values:

- $u, v$ : components of velocity
- $T$ : the temperature
- $S$ : the salinity of ocean water
- $\zeta$ : sea surface elevation
Introduction – Cost function

- Reduce $J(x)$ by seeking $x_{obs}$ at locations that will lead to the greatest amount of error reduction
Introduction - AUVs

- AUV – Autonomous Underwater Vehicle
Introduction - AUVs

- Advantages
  - Many scientific payloads in a small chassis
    - Acoustic Doppler Current Profilers
    - Temperature and Salinity sensors
  - Quick deployment
  - Capable of visiting places humans are not
Introduction - AUVs

- Disadvantages
  - Battery powered (limited mission time)
  - Limited computing resources
  - Can be fragile
Introduction

- Leverage advantages of the AUV against disadvantages to get the best reduction in ROMS
Accomplishments

- Avila Beach Model
- Optimization problem
- Java Simulation
Avila Beach Model

- Approximately at the center of given MATLAB model
  - Set up by Dr. Paul Choboter
- Represents a cubic rectangular area
  - (0,0, -497) feet at bottom left
  - (63820, 127700, -.125) at upper right
Avila Beach Model
ROMS Image – Side View

Vertical section of sigma coordinates, \( j \)-index = 120
ROMS Image – 3-D view
Objective

Optimization problem:

\[
\begin{align*}
\min J \\
\text{by finding } x_{\text{obs}} = \begin{bmatrix} x_{\text{obs}_0} y_{\text{obs}_0} \\ x_{\text{obs}_1} y_{\text{obs}_1} \\ \vdots \\ x_{\text{obs}_{tf}} y_{\text{obs}_{tf}} \end{bmatrix}
\end{align*}
\]

In \( t_f \) timesteps such that the routes in \( x \) result in the greatest drop in \( J \)
Objective

... subject to:

\[ J = \sum_{i,j} f_{i,j} \]

\( x_i \in [0, i_{\text{max}}] \quad l = 0...l_{\text{final}} \)

\( y_l \in [0, j_{\text{max}}] \quad l = 0...l_{\text{final}} \)

\( x_i - x_{i-1} \in [-1, 0, 1] \quad |x_l - x_{l-1}| + |y_l - y_{l-1}| = 1 \)  \( f \) is the error at \((x, y)\)

\( y_i - y_{i-1} \in [-1, 0, 1] \quad f_{i,j} = f_{i,j_{i-1}} + \nabla f(x_{l-1}, y_{l-1}) \)
Approach

- Simulation written in Java
  - OceanPoint, OceanMap, Node, and MapSearchTests classes
- OceanPoint
  - (x,y) coordinate + amount of error
- OceanMap: Discrete 2-D array of OceanPoints
- MapSearchTests
  - Breadth-, Depth-, and Best-first search methods
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;err&gt;</td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
</tr>
</tbody>
</table>

X

Y
Approach

- Want to visit as many different points as possible that result in the greatest reduction in error

- Constraints
  - Start and stop at the same node
  - Can only move Manhattan distances (forward-back or side-to-side)
  - Limited number of points can be visited
    - (Battery or time constraints, for example)
Solution

- Using Breadth-first and Depth-first:
  - Find all paths that start and finish at the same node
    - eliminating paths with duplicate points
    - in a predetermined number of steps (even only)
  - Serially analyze each path to determine how much that route reduces error in the OceanMap
  - Take the best!
Breadth-first

- Uses FIFO (First-in-first-out) queue to store nodes to be analyzed/searched
- Looks and stores nodes in a clockwise order (NESW)
  - Stores all nodes (puts them in the queue) BEFORE looking at the next one
- Memory-intensive because it stores ALL nodes to be searched in memory
Breadth-first

- Algorithm:

  **BreadthFirst**(goal, map, node, queue, pathList):
  
  if(node.stopsLeft == 0 and node.location == goal) then:
    add node to pathList
  else
    for each child of node: //in N-E-S-W order
      if(child within bounds of map)
        child.stopsLeft = node.stopsLeft -1
        add child to queue
    if(queue not empty) then:
      **BreadthFirst**(goal, map, queue.dequeue, pathList)
Breath-first Visual

```
<table>
<thead>
<tr>
<th>Add-1st</th>
<th>Add-2nd</th>
<th>Add-4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

X

Y
Breadth-first Queue

After 1 iteration:

(4,5), (5,4), (4,3), (3,4)
Breadth-first Queue

After 2 iterations:

(5,4), (4,3), (3,4), (4,6), (5,5), (4,4), (3,5)
Breath-first Visual
Breadth-first Queue

After 3 iterations:

(4,3), (3,4), (4,6), (5,5), (4,4), (3,5), (5,5), (6,4), (5,3), (4,4)
Depth-first

- Uses LIFO (Last-in-first-out) stack to store nodes to be analyzed/searched
- Looks and stores nodes in a clockwise order (NESW), like Breadth-first
  - BUT, looks and stores only at the first node (North, if available) before iterating again
- Less memory intensive because it can forget a path once that branch has been visited
Depth-first

Algorithm:

DepthFirst(goal, map, node, stack, pathList):
    if(node.stopsLeft == 0 and node.location == goal) then:
        add node to pathList and pop stack
    else
        for each child of node: //in N-E-S-W order
            if(child within bounds of map) then:
                child.stopsLeft = node.stopsLeft - 1
                push child onto stack
                DepthFirst(goal, map, stack.peek, stack, list of paths)
        pop stack
# Depth-first Visual

![Grid Diagram with Depth-first Search](image-url)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;err&gt;</td>
<td></td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;err&gt;</td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td></td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>Add-1st</td>
<td>START</td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td></td>
<td></td>
<td>&lt;err&gt;</td>
</tr>
<tr>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
<td>&lt;err&gt;</td>
</tr>
</tbody>
</table>

**X** and **Y** axes represent the grid's dimensions.
Depth-first Stack

After 1 iteration:

(4,5)
Depth-first Visual
Depth-first Stack

After 2 iterations:

(4,5), (4,6)
Depth-first Visual

```
<err>      <err>      <err>      <err>
<err>      <err>      <err>      <err>
<err>      <err>      START     <err>
<err>      <err>      <err>      <err>
<err>      <err>      <err>      <err>
<err>      <err>      <err>      <err>
```

X

Y
Depth-first Stack

After 3 iterations:

(4,5), (4,6), (4,7)
Solution

- **Using Best-First**
  - Expands the node that offers the best reduction in error for the model for that time step
  - More realistic – online search
    - Can react to a model that updates every time step
    - Doesn’t need to create a trajectory and then follow it – on-the-fly
    - However, lacks in that it only looks one ahead
Best-first Visual

![Diagram of a best-first search algorithm, showing the grid with values and arrows indicating the path from the start to the end.]
Video Demo

- Video shows map being updated with less error for each time step
Accomplishments - Limits

- Computer on AUV not capable of exploring a search space larger than ~10 x 10 using aforementioned search methods
  - Greedy best first avoids this problem, but does not yield the best solution
Current work

- Map resolution reduction
  - Merge a number of adjacent points into a single square with a higher error, but smaller resolution
Current Work - Example
**Current Work - Example**

```
  |   |   |   |
--+---+---+---+
  |   |   |   |
--+---+---+---+
  |   |   |   |
--+---+---+---+
  |   |   |   |
```
Current Work - Example
Concerns/Future Work

- None of the aforementioned solutions consider a time-dependent model
  - Map viewed as static; unchanging
- “Best-first” is not “Best”
  - Does not look ahead more than one
- What is the best resolution reduction factor?