Search Lab

September 20, 2004

The purpose of this lab is to give you experience with various kinds of search. You will code up several of them and see a graphical representation of how they work in the BZFlag world. Specifically, you will do the following:

- Code up Depth-First Search
- Code up Iterative Deepening Depth-First Search
- Code up Breadth-First Search
- Code up Uniform Cost Search
- Code up Greedy Search
- Code up $A^*$ Search
- Report on the lab

1 What to Turn In

To pass off this lab, you will turn in the following electronically:

- A declaration of time spent by each lab partner
- A lab report

You will also need to demonstrate to the TA that your code works.

This report should be more thorough than the last. Go through all of your experiments in such a way that they could be replicated by someone reading your paper. If you have had experience writing conference papers before, make your paper look like one of those, including an abstract, lab context, and details about what you tried.

Make sure that your report includes information like what heuristics you used. Try things with inadmissible heuristics as well and report on the optimality of the resulting searches.

Specifically, you will need to include information about the following:

- What heuristics were used along with the results of their application
- What happens when an inadmissible heuristic is used (give concrete results)
- How you would apply the search techniques in an online BZFlag environment, specifically:
  - How you would discretize the world
  - How you would measure edge costs
  - What your action space would be
  - What heuristic(s) you would use

Perhaps most importantly, put your results into a format that is easy to understand at a glance. Sometimes that means that you want to put things into a graph, and sometimes it doesn’t. Be creative, and consider that your audience wants to get maximum understanding with a minimum of effort. As usual, we suggest that you use $\LaTeX$ if you can.
2 Implementation Requirements

You will be given a distribution of BZFlag code. Included in the distribution are some files that you get to fill in (these are not part of the standard BZFlag distribution, so make sure that you get it from the class website).

You will implement several different searches to find a path from your current position to the enemy flag. The searches you will implement are the following:

- Depth-First
- Iterative Deepening DFS
- Breadth-First
- Uniform Cost
- Greedy
- A*

For each search, you will draw lines when you expand a node (showing which children will be considered) and when you pop a node. You will also draw lines in the world indicating which path is ultimately chosen as the shortest path. To do this, you will use functions provided for you as described below.

The code will be tested using the empty world. Your code should not have any magic numbers in it for queue sizes, etc. Grading will be done on the basis of final path costs (where applicable), memory and CPU usage (just make sure it isn’t out of control), and the correctness of the number of expanded nodes. Your code should perform good memory management (free allocated memory at the end of a search). Also, A* and Uniform Cost should give equal path costs, but A* should expand less nodes.

This distribution is different from past distributions, as it has modifications specific to the display of search paths. Make sure you get the right distribution. Put it into a place distinct from other distributions and do a full recompile, or you may have weird errors. To do a full recompile, change to the root directory of the distribution, and do the following:

```
./ configure
make
```

2.1 Lab API

The following functions constitute the majority of your API. Most of them should be self-explanatory. For all functions requiring a dir parameter, it should be one of the DIR_ constants defined in the top of SearchAgent.cxx. (Typical code should loop over all directions, from 0 to NUM_DIRECTIONS - 1).

```c
float get_path_cost( map_t *map, int x, int y, int dir );
int can_move_direction( map_t *map, int x, int y, int dir );
int is_enemy_flag( map_t *map, int x, int y );
int locate_enemy_flag( map_t *map, int x, int y );

void draw_link_to_parent( map_t *map, node_t *n );
void draw_goal_path( map_t *map, node_t *goal );
void int_drawline( map_t *map, int x1, int y1, int x2, int y2,
                   float r, float g, float b );
```

Make sure that your code draws the progress of the search frontier. To do this, simply draw a line from the parent node to a child node when it is expanded.

Each node_t struct has a parent member. As you expand nodes, you should make sure that the parent members are correctly updated. If you do this, you will automatically form a chain of node_t structures
which go from finish to start. That will enable you to use the draw\_goal\_path function provided when your
search terminates. It will also allow draw\_link\_to\_parent to work correctly when you hand it a child node.

Your search functions should return a pointer to the goal’s node\_t structure, and your code in the state\_2
function should set that return value equal to the goal variable.

2.2 Starting Things Up

- Build the Lab bots using SleeperAgent.cxx
  Rename the executable ta\_bot.
  (These agents just sit there, and provide something for the get\_path\_cost function to operate on. They
  use usleep() so they don’t spin the CPU too much.)
- Implement your search code in SearchAgent.cxx
  (or, for now, just soft-link and compile).
- Start up an observer
  ./observe.sh –geometry 600x600
  (NOTE: One of the major changes we have made is that the observer starts with mouse tracking OFF.
  Press F10 to toggle it).
- Start up the sleeping enemy agents.
  This will create a fun and interesting cost topology. Note that greenteam.sh automatically calls ta\_bot.
  ./greenteam.sh
- Run your agent. ./redteam.sh
  search\_type <depth|iterative|breadth|uniform|greedy|astar>

You may want to enable the cost topology (the line is commented out in state\_1) the first time you run
your code. It’s designed to help build your intuition regarding the cost function. Although it is very pretty
to look at, it does take some time to generate, so you’ll probably only want to use it once.

2.3 Discretizing the World

The BZFlag system operates in a continuous environment, but the search techniques you will implement
require a discrete environment. To save everyone some time, we have provided code which will discretize
the world for you. As long as you use the map\_t structure and the supporting functions we have defined, you
shouldn’t have any problems.

We strongly recommend that you read through all of the code provided in ‘SearchAgent.cxx’! It handles
a lot of things for you!

2.4 Drawing Lines in the Observer

There are three levels of abstraction to drawing lines. The easiest thing to do is to call the function

\begin{verbatim}
void draw\_link\_to\_parent( map\_t *map, node\_t *n );
\end{verbatim}

Assuming that you have set up the parent member of the node\_t structures correctly, this function will
automatically draw a red line from a node to its parent.

If you prefer more direct control, consider the following functions:

\begin{verbatim}
void int\_draw\_line( map\_t *map, int x1, int y1, int x2, int y2,
  float r, float g, float b );
void draw\_line( line\_t *l );
void clear\_lines( void );
\end{verbatim}
3 Important Notes

3.1 Other requirements

There is a total_nodes global variable which you should increment every time you expand a new node. This variable will be used to track the efficiency of your search method.

There is a search_type global variable which will inform you which of the search types is desired (it is tied to a command-line switch). Your code will be run at least once for each search type.

You will probably want to integrate some kind of search into your final agents, so try to write the search code in a general way that can be used later.

3.2 New Things

The observer has had some new keystrokes added. They are as follows:

- **F9** will clear all of the lines.
- **F10** will toggle mouse tracking (which is OFF by default)
- **F11** is Wingate’s hyper key. It will automatically take you to a bird’s eye view, toggle the black renderer, center everything, stop all mouse motion, and turn off mouse tracking.

There are also a few new worlds in the bzfs directory to look at. They’re very simple, but will hopefully help you test your code.

3.3 Code Issues

*It seems slooooooow:* The new system abuses the inter-player messaging facility to draw lines. Unfortunately, that has exposed some instabilities in the bzflag packet handling system. We couldn’t fix it, but noticed that if we slow things down, it tended to fix things. Of course, the instability is only manifest if zillions of inter-player messages are being sent, which is only a problem if you’re drawing lines. In the final lab, you won’t have to draw any lines, so it shouldn’t be a problem for anyone.

*Observer / green agents sometimes crash:* This sometimes happens due to the previous issue. It generally doesn’t affect the red agent, however, so you shouldn’t have any problems developing.

*The discretization isn’t perfect:* There is a slight problem with the world discretizer. Sometimes, a square building will occupy more grid squares than it should. It tends to have a small "tip" added at one of the corners:

```
..................
..................
...oooooooooooo...
...oooooooooooo...
...oooooooooooo...
...oooooooooooo...
...oooooooooooo...
..............o... <-- like this
..................
```

If your search seems to avoid certain grid squares, you may wish to use the print_map function to determine whether it’s the discretizer, or your code.