Discussion slides for week of November 26, 2007
The University of Michigan
Agenda

- Any Questions?
- HW4 (Due coming Tuesday)
- PA3 Questions/discussion
  - Last time
    - Branch & Bound
  - Today
    - Input parsing, “tokenizing”
    - PA3 review (BnB)
    - 2-approximation
    - k-Optimization, getrusage
    - Anytime algorithms
Parsing Input

- This is a pain, especially for complex formats

- You’ve already done this for PA1; much of your solution should translate to PA3

- A very common problem is extracting chunks of information from a delimited string
  - “Tokenizing” a string
  - Extract ints from “1 2 3 40 5 12” or “1,2,3,40,5,12”

Data Structures and Algorithms
Options for Tokenizing

- The C `strtok()` function

```c
char str[] = "1,2,3,40,5,12" ;
char *tmp;

printf ("Splitting string \"%s\" into tokens:\n",str);
tmp = strtok (str,"","");

while (tmp != NULL) {
    printf ("%s\n",tmp);
    tmp = strtok (NULL, ",");
}
```
Options for Tokenizing

- Use string library functions

```c++
string tmp;
string delims = " ", ";
int j = 0;
int i = s.find_first_not_of(delims, j);

while (i != string::npos) {
    j = s.find(delims, i);
    tmp = s.substr(i, j-i);
    I = s.find_first_not_of(delims, j);
    // process tmp
}
```
Options for Tokenizing

- Use the iostream get/getline functionality
  
  ```cpp
  istream& getline (char* s, streamsize n, char delim );
  ```

- For example:

  ```cpp
  char delim = ',';
  char buf[256];
  ifstream file("in.txt");
  while (file.good()) {
    file.getline(buf, 256, delim);
    // process token (stored in buf)
  }
  ```
Options for Tokenizing

- Use a stringstream (if the tokens are spaces)

```cpp
int i;
stringstream a("1 2 3 40 5 12");
while (a >> i) {
    // process i
}
```
PA3 – first task

• Read in the map.
  – The IO part of this shouldn't be too much different from what you've done before in the previous Pas.

• You must determine a good representation for your graph.
  – Weighted adjacency matrix
  – Adjacency list
  – Others...?

• Recommendation: create an interface for your graph that abstracts away the internal details.
  – That way, you can concentrate on the task at hand rather than the details
  – Also, should you choose to change your internal representation, in theory you would need to only change the class itself. Everything else could stay the same.
PA3 – first task

- By *interface*, I mean a common set of (public) methods on a class that will be consistent *regardless of implementation*.
  - We define the interface
  - Could be implemented with subclasses/polymorphism
  - Or, we could just agree to only access the public interface methods, and leave the internals to the class itself (this makes a lot of sense).
PA3 – Representation

- What kind of interface might we use for a graph representation?
  - *(There are potentially many possibilities)*

- Later, we might find that changing the interface is unavoidable. Sometimes this happens, but if we can minimize how often it happens, we're still better off.
Class Graph {
public:
    Graph();
    virtual ~Graph();

    Vertex& getVertex( unsigned int vtxIdx ) const;
    bool addVertex( unsigned int vtxIdx );
    bool addEdge(
        unsigned int vtxIdx0,
        unsigned int vtxIdx1,
        double weight );
};

class Vertex {
public:
    Vertex();
    virtual ~Vertex();

    LinkedList< Edge > getNeighbors();
...
};
PA3 – Generating tours: a naïve approach

- As a first pass, given a graph $G$, can we think of a naïve approach to generate tours?
  - Do a permutation approach.
    - For each permutation, if it is valid in $G$, then we measure its cost.
    - Return the lowest-cost cycle.
  - Observe that this permutation search is generating a kind of tree.
    - Each path from the root to a leaf represents a complete tour. Thus if there are $n$ leaves in the tree, there are $n$ tours.
    - We are searching all possible tours
PA3 – Branch and Bound

- If, for example, we encounter a path that we can determine will *not* yield an optimal tour (or maybe no more optimal than the best we've already seen), we can *prune* this branch of the search tree.

- This is the essence of Branch and Bound.
PA3 – 2-approximation

- In a nutshell, construct an MST for a graph.
- MST should be constructed using Prim's algorithm
- Need a priority queue (implemented using minHeap)
  - Does this resemble PA2? Remember…..code reuse!
- Traverse the MST (e.g., depth-first)
- Return the generated traversal
PA3 – k-Approximation

- From professor’s lecture notes:
  - Start with an initial tour, e.g., the randomly generated one or the MST
  - Consider two cities $v_i$ and $v_{i+1}$ or any two cities $u$ and $v$
  - Check if visiting $v_{i+1}$ before visiting $v_i$ decreases total tour length
  - Repeat with different $v_i$ until there is no further improvement (or up to time limit)

- Time limit of 60 seconds!

- You **must** use `getrusage(2)` to determine both the elapsed time and time remaining

- Use the timing information provided by this function to help regulate your heuristics and algorithms.
double getcputime(void) {
    struct timeval tmpTime;
    struct rusage ru;
    getrusage(RUSAGE_SELF, &ru);

    tmpTime = ru.ru_utime;
    double t = (double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;
    tim = ru.ru_stime;
    t += (double)tim.tv_sec + (double)tim.tv_usec / 1000000.0;

    return t;
}

Anytime algorithms

- In many applications, we need *an* answer, even if it is not the best answer
  - Available time may be unknown
  - Solution times can vary *widely* for different instances

- We want an algorithm that will always return the best answer it can, given the time we allow it
  - “Real time” or “anytime” computation
Solution Quality

![Graph showing the relationship between time and solution quality, with an optimal value plateau.

The graph illustrates the increase in solution quality over time, reaching an optimal value that remains constant.

The x-axis represents time, while the y-axis represents quality. The line on the graph shows the progression of solution quality as time increases, eventually plateauing at the optimal value.

This graph is likely from a lecture or presentation on data structures and algorithms, indicating the importance of understanding the trade-off between time and quality in algorithm design.
For PA3, you must return the best solution you can on 60 seconds. How could you approach this?
General Strategy

- Always store the best known solution to return when time is up (and for use as a bound)
- First, compute quick solutions that may not be very good
  - Random
  - Heuristic solution (e.g. greedy)
  - 2-approximation
- Local search for improvements (k-opt)
- Compute the optimal solution (BnB)
That’s it for today!

Any questions?