CMSC 420 Term Project

Hazardous Planet Annexation*

March 15, 2000

1 The Problem: Establishing a Base on Planet Temas

The infamous alien Sametists recently discovered a new planet, named Temas after their founder, and plan to colonize it. This project will guide you in finding the resources of this planet and helping colonists to access the resources needed for survival, while minimizing the labor and material cost of construction, without violating any of the physical constraints associated with the site. Note that *non-functional requirements*[1] such as the need for real-time performance demands that the code be efficient, as well as correct, as repair is both dangerous and costly.

The ulterior motive of this project is to illustrate the principles of combining data abstractions based on mixing and matching specific properties or attributes of the data types to produce composite data-structures that facilitate providing the required service to the customer. In this project, information stored in both linear and spatial data structures must be combined to solve various real-world-type problems.

The assignment is to be done in three segments, with the parts (and tentative due dates) specified below. Part 1 asks you to construct a *BST* to contain resource site names; to construct a *Point-Region quadtree* to represent the spatial relationship among the resources; and to build a command interpreter to be used in this and later parts of the project. In part 2, a *PM1 quadtree* of sites corresponding to a network of inhabitable bases, will be constructed and traversed to plan the introduction of colonists to the planet. In part 3 all these data structures will be used in developing a habitable planetary site.

1.1 Part 1: Minimal Resource Network Due: Mar. 8 Max. points: 150

This section describes the first portion of the project - the BST implementation for storing the site names; the PR quadtree to capture the site location; and, the adjacency-list implementation to store the route information. You will also build a command decoder to support commands for Part 1; you will expand it appropriately later to accommodate commands required for future parts. Each command spans exactly one line; commands will be in uppercase, and reasonably sloppy syntax is to be supported (spaces and empty lines are allowed). The commands to be supported for part 1 are as follows:

*Participation in this project may prove HAZARDOUS to your health. Unfortunately, failure to participate, will definitely have an adverse effect upon your GPA. Take my advice. Start now, because you're already behind.

[1]This is a little sarcasm on my part. Requirements that do not specify a function (action) of a controlled object, such as a battleship or a life-critical system component, are often referred to as *non-functional requirements*, even though they may require additional component functions.

- CLEAR_ALL() *[5 points]* initializes all data structures used in your program. This will always be the first command in the data set; it could also appear in the middle of a run which should then cause deletion of all currently defined sites and routes.
- CREATE_SITE(*site_name* , x, y) *[10 points]* adds the name of the site along with its coordinate position into the data dictionary. The data dictionary should be implemented as a BST ordered in the natural ASCII order (ie. strcmp() order) of the site names, with 'less than or equal to' values being inserted into the left subtree and 'greater than' values being inserted into the right subtree. Site names will be composed of 6 characters that are _ or alphanumeric. Coordinates will be in the range [0, 1024). Print confirmation message if this command is processed successfully; an error message otherwise. Note that creating a site that exists already is an error.
- PRINT_BST() *[15 points]* prints the inorder traversal of the BST as a listing of the site names and their coordinates. This function will be used as a measure of success for the CREATE_SITE function. Print an error message if the tree is empty.
- INSERT_SITE(*site_name*) *[20 points]* inserts the specified site into PR quadtree with left and bottom boundaries closed and right and top boundaries open. The site to be inserted should have been created earlier using CREATE_SITE command. If the site is not present in the data dictionary, output an error message. Print confirmation message if the command is processed successfully. If the site exists in the quadtree already, output an error message.
- PRINT_PRTREE() [25 points] prints the output of traversing the subtrees of the PR quadtree in the order NW, NE, SW and SE. When a leaf node is traversed the site name associated with it should be printed; whereas when a nonleaf node is traversed, the direction 'NW', 'NE', 'SW' or 'SE' should be printed. Output an error message if the PR quadtree is empty.
- CREATE_ROUTES(*site_name*) *[20 points]* constructs a route between the specified site and its closest neighbor in the PR quadtree, based on Euclidean distance. This function creates a minimal resource network connecting resource sites, and is implemented using adjacency lists associated with each site. Print all the routes existing in the database along with their lengths if the command is processed successfully; otherwise print an error message.
- RECTANGLE_SITES($x1, y1, x2, y2$) *[25 points]* identifies and prints all sites within a block specified by the endpoints of a diagonal of a rectangle. Print an error message if no sites are found. You are expected to use the PR quadtree efficiently to guide your search process. For full credit, your output should also include the path that you follow during the search. The path can be printed in terms of the nodes visited (leaf: names, nonleaf: direction. ref. PRINT_PRTREE()). Programs that do not prune the search tree efficiently will be penalized.
- RADIUS_SITES(*site_name* , *radius*) *[20 points]* identifies and prints all sites within the specified radius of a given site. Print an error message if no sites are found. You are expected to use the PR quadtree efficiently to guide your search process. For full credit, your output should also include the path that you follow during the search. The path can be printed in terms of the nodes visited (leaf: names, nonleaf: direction. ref. PRINT_PRTREE()). Programs that do not prune the search tree efficiently will be penalized.

There are 10 points allotted for program documentation, conformance to requirements and efficiency. So please make sure that you follow all instructions specified in this document, the webpage, the newsgroup and in the class.

1.2 Part 2: Minimum Spanning Tree and PM Quadtree Due: Apr. 10 Max. points: 150

In this part of the project you will expand your BST data structure from Part 1 to store the type of the site along with the existing information. The sites are classified into two classes: bases and raw materials. The bases are of types labs, barracks or supply. The raw materials are either power or minerals.

You will implement a PM1 quadtree (with quadrants closed on all sides) to store the routes between bases. The raw material sites are not directly connected to other sites. However the paths connecting the bases could pass close to or even through the raw material sites.

You will also compute a minimum spanning tree for all the base sites and identify all connected components.

- CLEAR_PM() *[5 points]* initializes or re-initializes a PM1 quadtree data structure. The spatial extent of both the coordinates is in the range of [0, 1024). Note that the PM1 quadtree should also be re-initialized if the command CLEAR_ALL is encountered.
- CREATE_SITE($site\_name, x, y, type$) *[10 points]* adds the name of the site along with its coordinate position into the data dictionary. The data dictionary should be implemented as a BST ordered in the natural ASCII order (ie. strcmp() order) of the site names, with 'less than' values being inserted into the left subtree and 'greater than' values being inserted into the right subtree. Site names will be composed of 6 characters that are _ or alphanumeric. Coordinates will be in the range [0, 1024). Type will be a single character with the following interpretation: 'L' - lab, 'B' - barrack, 'S' - supply, 'P' - power and 'M' - mineral. Print confirmation message if this command is processed successfully; an error message otherwise. Note that creating a site that has the same name or coordinates as an existing site is an error.
- LIST_SITES() *[10 points]* prints all base sites in ASCII order of names along with their types followed by the raw material sites in ASCII order of names along with their types. Print an appropriate error message if there are no base sites or raw material sites.
- INSERT_ROUTES($site\_name1$, $site\_name2$) *[20 points]* inserts an edge with endpoints $site\_name1$ and $site\_name2$ into the PM1 quadtree which has its quadrants closed on all sides. Both the specified sites should be valid site names; output an error message otherwise. If the command is called for an existing route, output an error message. Note that while inserting edges into the PM1 Quadtree, there exists a small possibility that two edges intersect at some point other than endpoints; if this happens, normally the recursive quadtree subdivision will loop infinitely (because there is no way to separate routes into different voxels). You should detect this situation by, controlling the recursion depth or current voxel size (or any other method) and if this happens, print an error message to indicate that the edge cannot be inserted and re-initialize the PM1 Quadtree (by deleting all nodes and edges in it) using an implicit call to CLEAR_PM. Print the created route as well as its length, if the command is processed successfully.
- LIST_ROUTES() *[15] points]* lists all existing routes in the PM1 quadtree along with their lengths. The routes should be listed in ASCII order, with each route being listed only once. (For eg. all routes from STE_01 will be listed before routes from STE_02. If there is a route between STE_01 and STE_02, that route will be listed as $STE\_01 <--> STE\_02$ and not $STE\_02 <--> STE\_01$.) Print an error message if the PM1 quadtree is empty.
- PRINT_PMTREE() *[20 points]* prints the output of traversing the subtrees of the PM1 quadtree in the order NW, NE, SW and SE. When a leaf node is traversed the type of information it contains 'V' - vertex, 'Q' - Qedge or 'N' - Null should be printed; whereas when a nonleaf node is traversed, the direction 'NW', 'NE', 'SW' or 'SE' should be printed. Output an error message if the PM1 quadtree is empty.
- PRINT_CC() *[20 points]* prints all maximal connected components of base sites, by computing the MST of the graph of base sites. Output should include the routes listed in terms of base names and total length of each connected component as the sum of individual route lengths. Your output should also include the total number of connected components. Print an error message if there are no base sites in the database.
- NEAREST_BASE($site\_name$) *[20] points]* finds the nearest base site to the given raw material site in terms of Euclidean distance. Print an error message if $site\_name$ is not a raw material site or if there are no base sites. If successful, output should include the base site name, type and its xy-coordinates. If two or more base

sites are equidistant from the given raw material site, print the name of any of those. Your search should be efficient. As a measure of efficiency, you should print out the paths that you are following in your PM1 quadtree (as you did in Part1). Programs that do brute-force search through the entire tree will be penalized.

- NEAREST_ROUTE($site\_name1, site\_name2$) *[20] points]* finds the route that is nearest to the given raw material site in terms of Euclidean distance. Output an error message if $site\_name1$ is not a base site or $site\_name2$ is not a raw material site. Print an error message if there are no routes in the database. If successful, output should include the route and its length. Your search should be efficient. As a measure of efficiency, you should print out the paths that you are following in your PM1 quadtree (as you did in Part1). Programs that do brute-force search through the entire tree will be penalized.

There are 10 points allotted for program documentation, conformance to requirements and efficiency. So please make sure that you follow all instructions specified in this document, the webpage, the newsgroup and in the class.

2 Instructions and Policies

2.1 General Information

Your project must execute on the OITs cluster. Otherwise, it will not be graded. Your program will be compiled and executed by the TA on the OITs cluster, and grades will be assigned on the basis of this execution. Your executable should be named part# [**Note: everything in lowercase**] in the makefile, where # must be replaced with the appropriate part number ie., 1 for part 1.

Here is an *example* of a makefile that would create an executable part2 in c++.

```
all:	part2.cc skiplist.cc skiplist.h
	g++ -o part2 -O2 part2.cc skiplist.cc -lm
```

Here is an example of a makefile for java

```
all:	part2.java skiplist.java Dijkstra.java
	javac *.java
```

The first line of the makefile should include all the files involved in your project; the second line is the compilation command itself. There should be TAB symbol after all: in the first line and at the beginning of the second line. To compile your program simply type "make".

2.2 Submission Instructions

You are required to submit your work electronically using the submit command. Follow the procedure outlined below to submit your project. Note that # should be replaced with the appropriate part number. (ie., for part 1, # should be replaced with 1).

1. mkdir part#

2. (a) **For regular submission:**
    Copy all the source code (.c, .cc, .java, makefile, .h) into this directory (part#). **Do not copy executables or object files.**

   (b) **For regrades:**
    i. For each file that has been modified, create a new file with the same base filename but with extension .diff using the diff command. For instance, if you modified skiplist.cc; execute

    ```
    diff skiplist.cc.old skiplist.cc > skiplist.diff
    ```

    ii. Copy all the .diff files into this directory (part#). **Do not copy any source code, executables or object files.**

3. cd part#

4. tar -cvf part#.tar *

5. gzip part#.tar

6. `~mh42060/lbin/submit # part#.tar.gz`

If you get any errors, report them to your TA immediately.

2.3 Late Policy

All projects are due at 22:00 on [some] Monday for being graded without any late penalty. The late penalties are as follows:

Projects submitted before	Late penalty (as % of max. score)
22:00 Monday	0
10:00 Tuesday	15
22:00 Wednesday	25
22:00 Friday	50
after Friday	100 (no credit)

No projects will be accepted for grading after 22:00 hours Friday. Please note that a project submitted later would overwrite the one submitted earlier. Therefore only the last submitted version before 22:00 hours Friday, will be graded.

2.4 Regrade Policy

You could submit the differences between your working version and the original version, for getting 50% of your lost credits (excluding the late penalty) back. (For example, suppose your original score was 60 (max. 100) which includes a late penalty of 10; you can make the project working fine and get 50% of 30 points back to get a total of 75.) Projects to be regraded are due before 22:00 on Friday in the week following the one in which the graded projects are returned.

3 Miscellaneous

Further details on the project will appear on the web page or in the newsgroup. You should check the web page regularly, as software specifications in natural language are inherently confusing and often incomplete. This specification will be augmented as needed.

Project related questions: contact darsana by email : darsana@cs.umd.edu
during office hours : TuFr : 10:00 a.m - noon or by appointment.