Grid World Design

Dennis Aurelian Leancu

gw\_build(int nrows, int ncols, int pop, int rnd)

Runtime: O(CR + N)

Design: Space must be allocated for a structure for every district. This is done in O(CR) time since there is no other way but to go through each district in the 2D grid. Additionally we must allocated space for a people structure. There is no other way but to do this for each person in the function parameter. Thus the final runtime is O(CR + N). This document outlines example structures for each district and person in the remaining functions below.

Example grid world structure (as a pseudo JSON object):

district[row][col] (array): {

population (int)

members (double linked list): {

id (int)

prev (pointer to members)

next (pointer to members)

}

}

people[id] (array): {

row (int)

col (int)

status (bool)

districtPointer (pointer to members)

}

availableIds (linked list stack): {

id (int)

next (pointer to availableIds)

}

peopleArraySize (int)

total\_pop (int)

maxId (int)

gw\_members(GW\* g, int i, int j, int\* n)

Runtime: O(Nij)

Design: Design a structure that contains the members for each district structure. The example we use will be a double linked list. Accessing the linked list is a constant time operation. Iterating through the linked list is an operation equal to the number of nodes in the linked list. Since by design, the linked list contains only the current persons in the district, the function is O(Nij).

Example code:

membersPointer = g->district[i][j]->members;

int array[g->district[i][j]->population];

while (membersPointer != NULL)

push membersPointer->id to array[];

membersPointer = membersPointer->next;

return array;

gw\_district\_pop(GW\* g, int i, int j)

Runtime: O(1)

Design: Use a data structure that allows constant time access across all indexes. For example, arrays provide constant time indexing. Population will need to be updated whenever a person moves or dies.

Example code:

return g->district[i][j]->population;

gw\_total\_pop(GW\* g)

Runtime: O(1)

Design: Use a data structure that allows constant time access within the main structure. Total population will need to be updated whenever a person dies.

Example code:

return g->total\_pop;

gw\_move(GW\* g, int x, int i, int j)

Runtime: O(1)

Design: Use a data structure that allows constant time access to a person’s data, then overwrite the current data with the new data within the people structure. Remember to update the population totals contained in the old and new district structures. Additionally remember to remove person from old district structure and add to new district structure. Removing a node in the District linked list is a O(1) operation if we store the pointer in the person structure. Adding a node to the head/tail of a double linked list is also a O(1) operation, therefore “gw\_move” is constant time.

Example code:

g->people[x]->row = i;

g->people[x]->col = j;

g->people[x]->districtPointer->prev =

g->people[x]->districtPointer->next; // Remove from old list

g->people[x]->districtPointer = addToDistrictList(i, j); // O(1) function that returns pointer to linked list for new district

gw\_find(GW\* g, int x, int \*r, int \*c)

Runtime: O(1)

Design: Use a data structure that allows constant time indexing given a unique identifier. As an example, arrays provide constant time indexing. A custom people structure can be used to hold information about each person. This function and function “gw\_move” are closely linked together by virtue of the people structure.

Example code:

if (g->people[x]->status)

\*r = g->people[x]->row;

\*c = g->people[x]->col;

gw\_kill(GW\* g, int x)

Runtime: O(1)

Design: Use a data structure that allows constant time indexing given a unique identifier. As the examples in the above functions described, an array can be used to contain a custom people structure. We can update their living status and district population totals in constant time. When a person is killed, their ID should be added to a structure containing available IDs for the next person created.

Example code:

g->people[x]->status = 0;

g->district[g->people[x]->row][g->people[x]->col]->population -= 1;

pushAvailId(g->availableIds, x);

gw\_create(GW\* g, int i, int j)

Runtime: Amortized O(1)

Design: Check if district given is valid (we are given the size of the grid world at start so this is trivial). Then check the list of available IDs. If there is an ID available, remove it from the list and assign it to the newly created person (this can be done in constant time if the list structure is a linked list functioning as a stack). However, if no previously used IDs are available, we need to create a new ID for this person. If we use the example pseudo structures described earlier, we must check if the people array structure has enough space for one more person. If there is room, we update the data structures to reflect a new maximum ID, otherwise we double the array size. This operation results in an amortized O(1) runtime.

Example code:

if (g->availableIds->id != NULL)

g->people[g->availableIds->id]->row;

g->people[g->availableIds->id]->col;

pop(g->availableIds);

else

if (g->maxId == g->peopleArraySize)

double the array size;

g->maxId++;

g->people[g->maxId]->row;

g->people[g->maxId]->col;

gw\_free(GW\* g)

Design: Since we created the grid world structure, we know which structures were dynamically allocated (and as such, the structures that need to be freed). Simply using the “free” function on every applicable structure is enough.