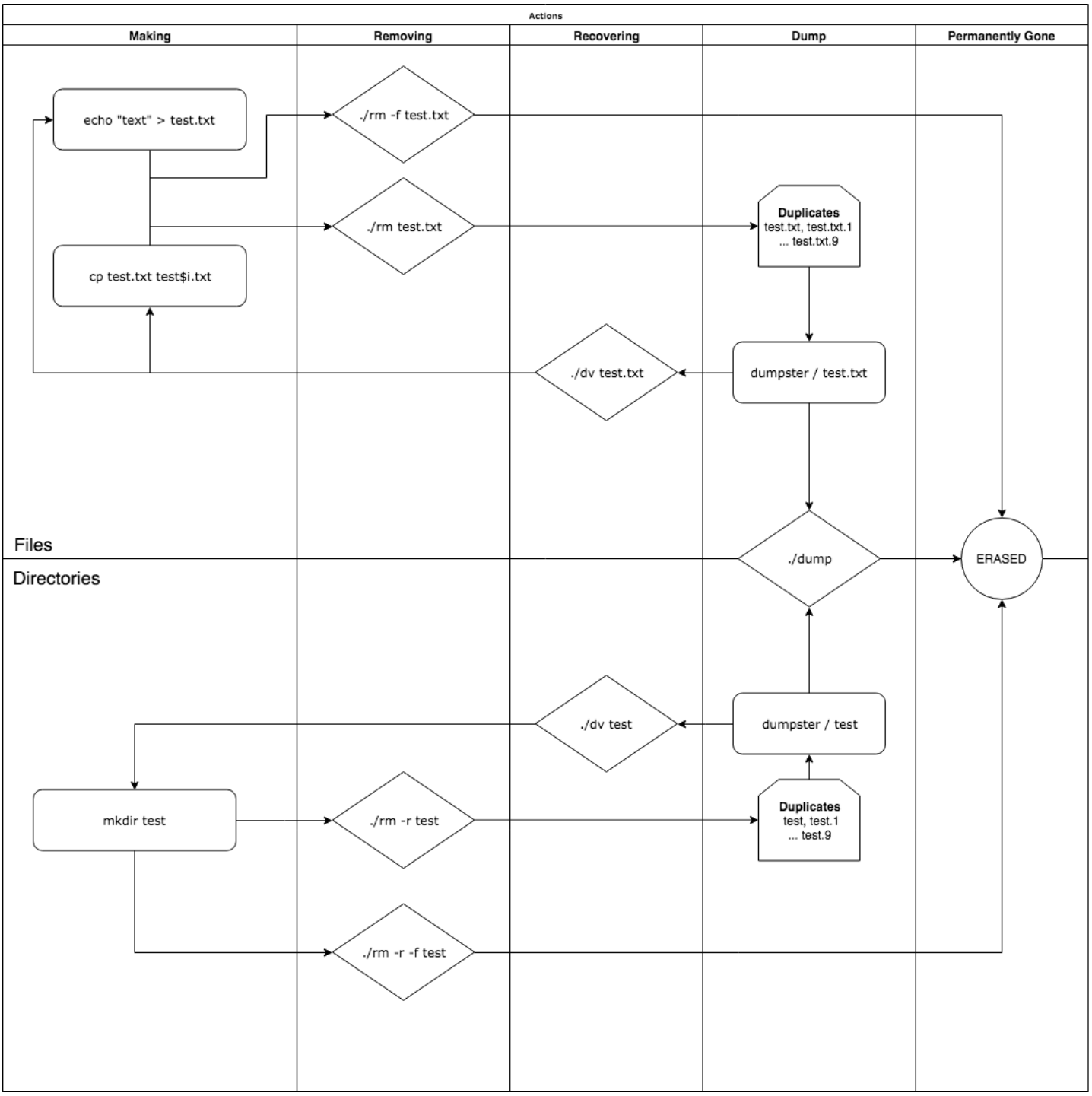
PROJECT 1: DUMPSTER DIVING

I created the Dumpster Diving Utility tool to remove files or directories either permanently or to a dumpster or trash. The user can then choose to dive into the dumpster to recover the file or dump the dumpster. If the user rm a file or directory already in the dumpster, it will add an extension .1, .2, … .9 to the name.



1. DESIGN

A. Programs / Scripts (pseudo-code)

i. Error Calls: checks for following constraints, if something goes wrong, ERROR message and prints usage message for each utility tool.

1. *ERROR_call(): default to usage message and exits*
2. *ERROR_no_file(int argc): checks if file or directory present*
3. *ERROR_rename_call(): checks if rename call fails*
4. *ERROR_remove_call(): checks if remove call fails*
5. *ERROR_open_call(): checks if open call fails*
6. *ERROR_opendir_call(): checks if opendir call fails*
7. *ERROR_mkdir(): checks if mkdir call fails*
8. *ERROR_duplicate_dir(): checks if directory already present*
9. *ERROR_rmdir_call(): checks if rmdir call fails*
10. *ERROR_limit_dumpster(): limit dumpster size to 10 same file or directory*
11. *ERROR_stat_call(): checks if stats are passed when moving*
12. *ERROR_chmod_call(): checks if permissions are passed when moving*
13. *ERROR_utime_call(): checks if utime call fails*
14. *ERROR_unlink_call(): checks if unlink call fails*
15. *ERROR_fileNotFound(char* source_path): checks if file is in dumpster*
16. *ERROR_rw_file(): checks if can read or write to file*
17. *ERROR_fopen_call(): checks if open file*

ii. Common: functions across two or more utilities

1. *concat(char* s1, char* s2): concate two strings*[1]
2. *flag_check(int arc, char** argv): check flag input*
3. *set_dumpster(): set the dumpster to be used to place files or directories*
4. *remove_force(char* directory): remove files or directories permanently*
5. *remove_directory(char* current_path, char* current_dumpster_path, int same): remove dir(s)*

iii. Rm: move file or directories to dumpster (same or different partition)

1. *usage(): prints usage message(with -h or error) "./rm -f, -h, -r < file(s) >*
2. *check_f_flag(char* file): check if -f flag is present*
3. *check_r_flag(char* file): check if -r flag is present*
4. *check_dumpster(): check to see where the dumpster is located*
5. *set_file(char* file): set the file to be moved*
6. *get_extension(char* path): get extension is duplicates*
7. *get_dumpster_path(char* file, char* dumpster_path, char** new_path): dumpster path*
8. *copyto_dump(char* file, char* dumpster_path, struct stat file_stat): move to dumpster*

iv. Dv: dive into the dumpter to recover files or directories

---

[1] https://stackoverflow.com/questions/8465006/how-do-i-concatenate-two-strings-in-c

1. *usage(): prints usage message(with -h or error) "./dv -h < file(s) >*
2. *copyto_target(char* source_path, char* current_target_path, struct stat file_stat): copies from dumpster to target*

v. Dump: cleans dumpster of all files and directories

1. *usage(): prints usage message(with -h or error) "./dump -h"*

B. Number of Runs (recording time in milliseconds)

I tested small and large files and directories. Small files were about 7 bytes and large files were about 7 megabytes. I ran these tests for same partition and different partition. The order of these actions are below:

> [Making 20 test files]

> [Removing 20 test files]

> [Recovering 20 test files]

> [Making 20 test directories]

> [Removing 20 test directories]

> [Recovering 20 test directories]

> [Removing 20 files and directories]

> [Emptying the Dumpster ☺]

C. Data Recording Method

In a script, "testing.sh", I created for-loops to create, remove and recover "test$i.txt" files and "test$i" directories. I used "sync" to make sure these operations were flushed to disk instead of cache. I got the initial and final times around each action to compute delta run speed in milliseconds. I either used "echo 'hello' > 'test$i' " (7 bytes) or "cp test.txt 'test$i.txt' " (7 megabytes) to copy a template file. I ran the command "./rm -r < fileOrDirectory… >" to move the file or directories to the dumpster (needing -r flag for directories. To recover files or directories, I ran "./dv < fileOrDirectory… >". I removed those files and directories again to the dumpster to then run "./dump" to clean the dumpster. I recorded all of these run times in milliseconds and can be found in the results section below.

D. System Conditions

I used Ubuntu Linux using the Virtual Box VM on my Macbook Pro. The version of Ubuntu is from the course website and my program works on that machine.

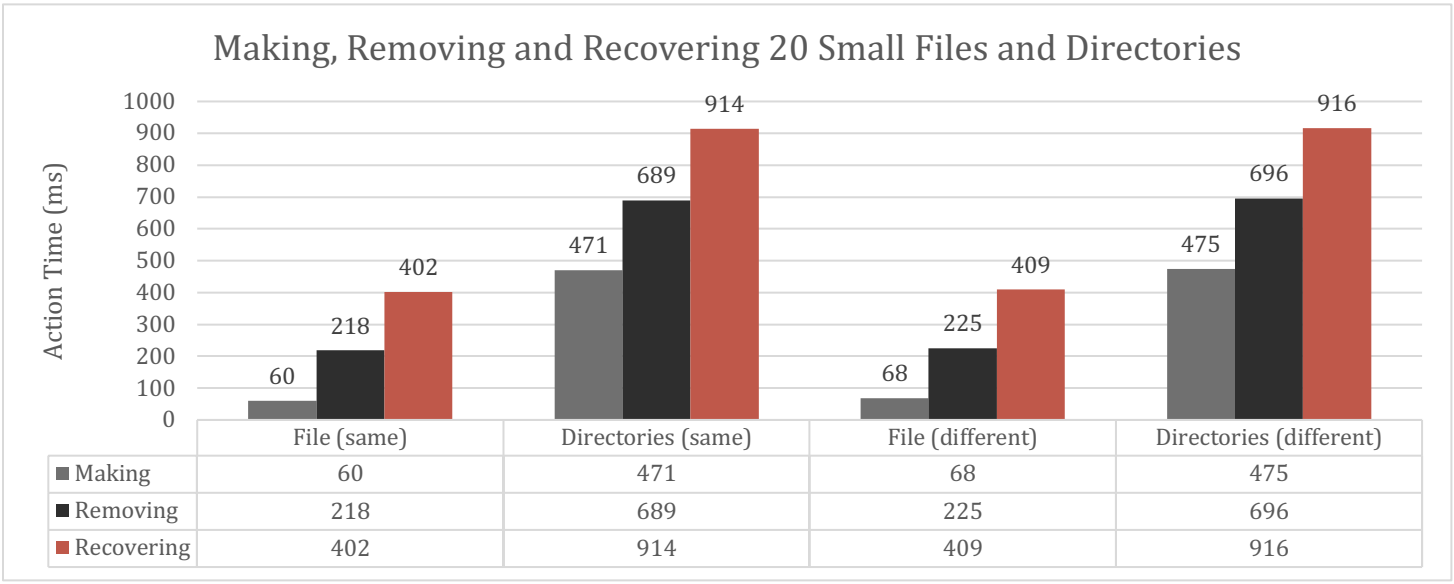
E. Other Details

My program has support for duplicate extensions on both files and directories. If a user removes a file or directory, there is a ".i" extension based on how many already in the dumpster. The limit is 10 duplicate files or directories (from 0 to 9). The user can use the -f flag to force remove a file and -r to recursively remove directories. Use -h for each tool to bring up the help usage message.

2. RESULTS

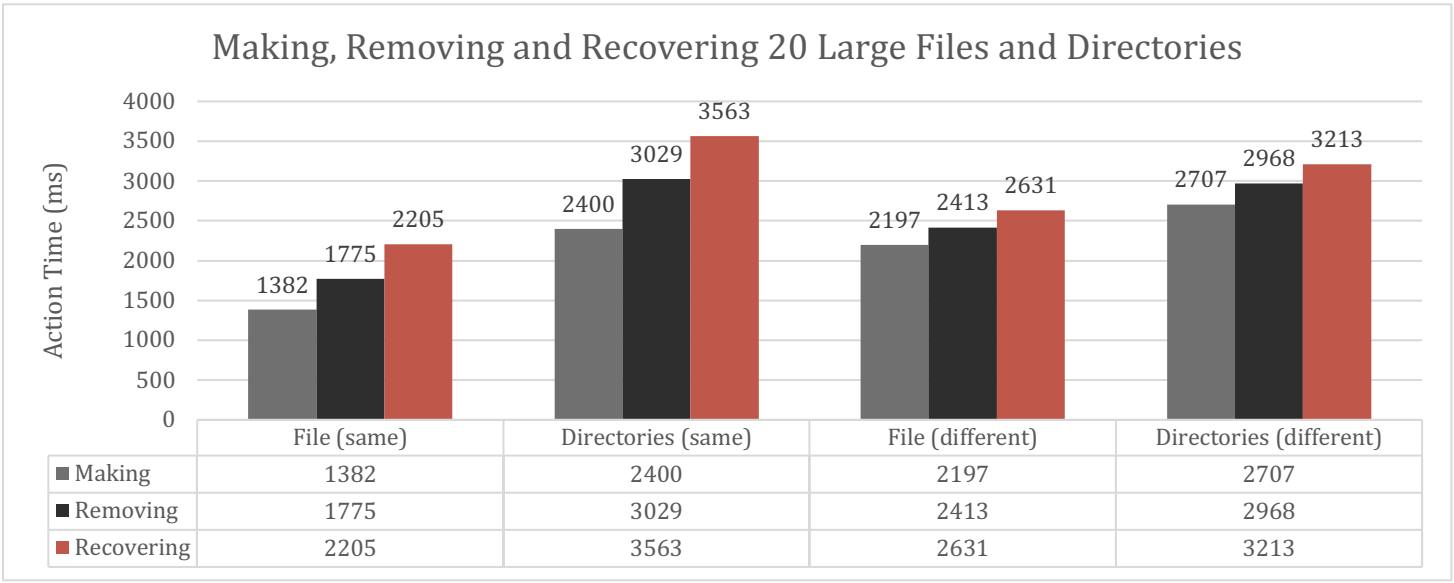
A. Tables or Graphs

I measured time for Making, Removing and Recovering files and directories. I measured different partitions, in my case parent folders on the virtual machine, to see if any changes in time. The first graph illustrates time to Make, Remove and Recover 20 (small 7 byte) files and 20 directories. The second graph measures time to Make, Remove and Recover 20 (larger 7 megabyte) files and 20 directories. Additionally, I tested for Rm of both files and directories, and Dumping the dumpster speeds.



	File (same)	Directories (same)	File (different)	Directories (different)
Making	60	471	68	475
Removing	218	689	225	696
Recovering	402	914	409	916

Rm Files and Directories	1195 milliseconds	1171 milliseconds
Dump	64 milliseconds	71 milliseconds



	File (same)	Directories (same)	File (different)	Directories (different)
Making	1382	2400	2197	2707
Removing	1775	3029	2413	2968
Recovering	2205	3563	2631	3213

Rm Files and Directories	3324 milliseconds	3524 milliseconds
Dump	89 milliseconds	116 milliseconds

B. Calculations (Mean and STD)

I tested 20 files and directories to be Made, Removed, Recovered, Removed again and then Dumped. I tested this for two different partitions for two different size files. I tested small 7 byte files and large 7 megabyte files. I averaged time to complete these actions to be 4,022 milliseconds for small files and 19,098 to larger files. Narrowing them down to per file and then to per byte, I have discovered smaller files compute at 28.71 milliseconds per byte vs. larger files compute at 0.0001363 milliseconds per byte.

TOTAL	**Small Files (n = 7 bytes)**	**Large Files (n = 7 megabytes)**
Same Partition	60 + 218 + 402 + 471 + 689 + 914 + 1195 + 64 = **4013**	2029 + 2220 + 2429 + 2501 + 2760 + 3024 + 3324 + 89 = **18376**
Different Partition	68 + 225 + 409 + 475 + 696 + 916 + 1171 + 71 = **4031**	2197 + 2413 + 2631 + 2707 + 2968 + 3213 + 3576 + 116 = **19821**
Average Time	**4022 milliseconds**	**19,098 milliseconds**
Per File (1 / 20)	**201 milliseconds**	**954 milliseconds**
Per Byte (1 / n)	**28.71 milliseconds**	**0.0001363 milliseconds**

3. ANALYSIS

A. Describe what the results mean

I can conclude that as the file size gets bigger, each byte is moved faster. Comparing the different partitions also gave a larger difference when the file size was larger. I believe this can mean that each byte of the file is being processed quicker when the file is larger. When the file is smaller, each byte is processed slower. My data is all relative based on my computer's CPU. I tested my experiment again on the same virtual machine on the same system and got different speed times based on how much CPU is being dedicated to the virtual environemnt.

B. How does your prediction match measured? If there is a difference, why do you think there is?

I have originally predicted that when the file size increases, time spent on each byte would decrease. I proved this by calculating time for each byte of small and large files. Small files (7 byte) would process each byte at 28.71 milliseconds. Large files (7 megabytes) would process each byte at 0.0001362 milliseconds.