Lab Exercise #4: Backtracking Search; Recursion

Andrew Crowell

The Solution

The following is the output from my recursive algorithm when supplied a 3L and 5L bucket and a desired water level of 4L.

```
overkill@overkill:~/CIS/CIS2520/Lab4$ ./TwoBucket
A = 0, B = 0
Fill A
A = 3, B = 0
Pour A into B
A = 0, B = 3
Fill A
A = 3, B = 3
Pour A into B
A = 1, B = 5
Empty A
A = 0, B = 5
Pour B into A
A = 3, B = 2
Empty A
A = 0, B = 2
Pour B into A
A = 2, B = 0
Fill B
A = 2, B = 5
Pour B into A
A = 3, B = 4
Empty A
A = 0, B = 4

Solution found!
```

How my Implementation Works

While it is not the best solution to this problem, it is a recursive solution that uses backtracking to good effect.

The recursive step of the implementation involves one (or all) of the following actions: pouring the first bucket into the second bucket, pouring the second bucket into the first, filling the first bucket, filling the second bucket, emptying the first bucket, emptying the second bucket. Each of these actions is followed by performing the rest of the actions needed to find a solution. If all combinations of actions fail, there is no solution (returns false). If the desired water level is obtained in one of the two buckets, and the other

bucket is empty, then the solution was found (returns true).

I got around cyclic steps by having a list of bucket states and action being performed. If there was a certain step that was going to be attempted with identical bucket volumes and identical action, it was apparent there was a cycle developing, so repeated steps would fail. If a certain combination of steps failed to yield results, it would undo the actions it performed and try again with different action combinations.

In order to vastly simplify the logic of the action-repetition detection, I have separated the bucket and action tracking operations into different files, and I use the list ADT that Dave provided with Assignment 1 to store actions. It is elegant and clean, and nice because my particular implementation is generalized for any combination of two buckets and any desired water level result. This is no insane person here, just a person who likes to design elegant code without hacky magic and fully-fixed frameworks. This code is actually quite simple, it's just perhaps too modular for some tastes (Not mine). With a little effort, this could solve the problem with three buckets, or N buckets, not that you'd *want* to or anything, but it'd be simple enough!

The Code

Behold the mighty source code!

TwoBucket.c

```
#include "adt/List.h"
#include "Action.h"
#include "Bucket.h"

bool TwoBucket(List* actions, Bucket* a, Bucket* b, int desired_level)
{
	if((BucketGetLevel(a) == desired_level && BucketGetLevel(b) == 0)
		|| (BucketGetLevel(a) == 0 && BucketGetLevel(b) == desired_level))
	{
		ActionPrintAll(actions, stdout);
		printf("A = %d, B = %d\n\n", BucketGetLevel(a), BucketGetLevel(b));
		return true;
	}
	else
	{
		if(BucketGetLevel(a) && !BucketIsFull(b))
		{
			if(!ActionWasPerformed(actions, a, b, POUR_A_TO_B))
			{
				PourAtoB(actions, a, b);
				if(!TwoBucket(actions, a, b, desired_level))
				{
					ActionUndo(actions, a, b);
```

```
            }
            else
            {
                  return true;
            }
      }
      else
      {
            return false;
      }
}
if(BucketGetLevel(b) && !BucketIsFull(a))
{
      if(!ActionWasPerformed(actions, a, b, POUR_B_TO_A))
      {
            PourBtoA(actions, a, b);
            if(!TwoBucket(actions, a, b, desired_level))
            {
                  ActionUndo(actions, a, b);
            }
            else
            {
                  return true;
            }
      }
      else
      {
            return false;
      }
}
if(!BucketGetLevel(a))
{
      if(!ActionWasPerformed(actions, a, b, FILL_A))
      {
            FillA(actions, a, b);
            if(!TwoBucket(actions, a, b, desired_level))
            {
                  ActionUndo(actions, a, b);
            }
            else
            {
                  return true;
            }
      }
      else
      {
            return false;
      }
}
if(!BucketGetLevel(b))
{
      if(!ActionWasPerformed(actions, a, b, FILL_B))
      {
            FillB(actions, a, b);
            if(!TwoBucket(actions, a, b, desired_level))
            {
                  ActionUndo(actions, a, b);
            }
```

```
                        else
                        {
                              return true;
                        }
                  }
                  else
                  {
                        return false;
                  }
            }
            if(BucketGetLevel(a))
            {
                  if(!ActionWasPerformed(actions, a, b, EMPTY_A))
                  {
                        EmptyA(actions, a, b);
                        if(!TwoBucket(actions, a, b, desired_level))
                        {
                              ActionUndo(actions, a, b);
                        }
                        else
                        {
                              return true;
                        }
                  }
                  else
                  {
                        return false;
                  }
            }
            if(BucketGetLevel(b))
            {
                  if(!ActionWasPerformed(actions, a, b, EMPTY_B))
                  {
                        EmptyB(actions, a, b);
                        if(!TwoBucket(actions, a, b, desired_level))
                        {
                              ActionUndo(actions, a, b);
                        }
                        else
                        {
                              return true;
                        }
                  }
                  else
                  {
                        return false;
                  }
            }
            return false;
      }
}

int main()
{
      List* actions = listCreate();
      Bucket* a = BucketNew(3);
      Bucket* b = BucketNew(5);
      int desired_level = 4;
```

```
      if (TwoBucket(actions, a, b, desired_level))
      {
            printf("Solution found!\n");
      }
      else
      {
            printf("No solution.\n");
      }

      BucketFree(a);
      BucketFree(b);
      listDelete(actions);
}
```

Bucket.h

```
#ifndef BUCKET_H
#define BUCKET_H
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct Bucket_t
{
      int capacity;
      int level;
} Bucket;

Bucket* BucketNew(int capacity);
void BucketFree(Bucket* this);
void BucketFill(Bucket* this);
void BucketEmpty(Bucket* this);
bool BucketIsFull(Bucket* this);
int BucketGetLevel(Bucket* this);
void BucketSetLevel(Bucket* this, int level);
int BucketGetCapacity(Bucket* this);
void BucketPour(Bucket* this, Bucket* dest);
#endif
```

Bucket.c

```
#include "Bucket.h"

Bucket* BucketNew(int capacity)
{
      Bucket* new = malloc(sizeof(Bucket));
      new->capacity = capacity;
      new->level = 0;
}

void BucketFree(Bucket* this)
{
      free(this);
}
```

```
void BucketFill(Bucket* this)
{
	this->level = this->capacity;
}

void BucketEmpty(Bucket* this)
{
	this->level = 0;
}

bool BucketIsFull(Bucket* this)
{
	return this->level == this->capacity;
}

int BucketGetLevel(Bucket* this)
{
	return this->level;
}

void BucketSetLevel(Bucket* this, int level)
{
	this->level = level;
}

int BucketGetCapacity(Bucket* this)
{
	return this->capacity;
}

void BucketPour(Bucket* this, Bucket* dest)
{
	if(dest->level + this->level <= dest->capacity)
	{
		dest->level += this->level;
		this->level = 0;
	}
	else
	{
		this->level -= dest->capacity - dest->level;
		dest->level = dest->capacity;
	}
}

int BucketPrint(Bucket* this, FILE* f)
{
	fprintf(f, "Bucket %d / %d\n", this->level, this->capacity);
}
```

Action.h

```
#ifndef ACTION_H
#define ACTION_H
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "adt/List.h"
```

```
#include "Bucket.h"

typedef enum ActionType_t
{
	POUR_A_TO_B,
	POUR_B_TO_A,
	FILL_A,
	FILL_B,
	EMPTY_A,
	EMPTY_B
} ActionType;

typedef struct Action_t
{
	int level_a, level_b;
	ActionType actionType;
} Action;

void ActionTypePrint(ActionType actionType, FILE* f);
void ActionRemember(List* actions, Bucket* a, Bucket* b, ActionType actionType);
void ActionUndo(List* actions, Bucket* a, Bucket* b);
bool ActionWasPerformed(List* actions, Bucket* a, Bucket* b, ActionType
actionType);
void ActionPrintAll(List* actions, FILE* f);
void PourAtoB(List* actions, Bucket* a, Bucket* b);
void PourBtoA(List* actions, Bucket* a, Bucket* b);
void FillA(List* actions, Bucket* a, Bucket* b);
void FillB(List* actions, Bucket* a, Bucket* b);
void EmptyA(List* actions, Bucket* a, Bucket* b);
void EmptyB(List* actions, Bucket* a, Bucket* b);

#endif
```

Action.c

```
#include "Action.h"

void ActionTypePrint(ActionType actionType, FILE* f)
{
	switch(actionType)
	{
		case POUR_A_TO_B:
			fprintf(f, "Pour A into B\n");
			break;
		case POUR_B_TO_A:
			fprintf(f, "Pour B into A\n");
			break;
		case FILL_A:
			fprintf(f, "Fill A\n");
			break;
		case FILL_B:
			fprintf(f, "Fill B\n");
			break;
		case EMPTY_A:
			fprintf(f, "Empty A\n");
			break;
		case EMPTY_B:
```

```
                fprintf(f, "Empty B\n");
                break;
            default:
                fprintf(f, "Unknown action!\n");
                break;
        }
}

void ActionRemember(List* actions, Bucket* a, Bucket* b, ActionType actionType)
{
        Action* action = malloc(sizeof(Action));
        action->level_a = BucketGetLevel(a);
        action->level_b = BucketGetLevel(b);
        action->actionType = actionType;
        listTail(actions);
        listAddAfter(actions, action);
}

void ActionUndo(List* actions, Bucket* a, Bucket* b)
{
        Action* action;

        listTail(actions);
        action = listDelCurrent(actions);

        BucketSetLevel(a, action->level_a);
        BucketSetLevel(b, action->level_b);
        free(action);
}

void ActionPrintAll(List* actions, FILE* f)
{
        List* iterator;
        Action* current;

        iterator = listHead(actions);
        current = listGetCurrent(actions);
        while (iterator != NULL)
        {
                fprintf(f, "A = %d, B = %d\n", current->level_a, current->level_b);
                ActionTypePrint(current->actionType, f);
                iterator = listNext(actions);
                current = listGetCurrent(actions);
        }
}

bool ActionWasPerformed(List* actions, Bucket* a, Bucket* b, ActionType
actionType)
{
        List* iterator;
        Action* current;

        iterator = listHead(actions);
        current = listGetCurrent(actions);
        while (iterator != NULL)
        {
                if(current->level_a == BucketGetLevel(a) && current->level_b ==
BucketGetLevel(b) && current->actionType == actionType)
```

```
            {
                  return true;
            }
            iterator = listNext(actions);
            current = listGetCurrent(actions);
      }
      return false;
}

void PourAtoB(List* actions, Bucket* a, Bucket* b)
{
      ActionRemember(actions, a, b, POUR_A_TO_B);
      BucketPour(a, b);
}

void PourBtoA(List* actions, Bucket* a, Bucket* b)
{
      ActionRemember(actions, a, b, POUR_B_TO_A);
      BucketPour(b, a);
}

void FillA(List* actions, Bucket* a, Bucket* b)
{
      ActionRemember(actions, a, b, FILL_A);
      BucketFill(a);
}

void FillB(List* actions, Bucket* a, Bucket* b)
{
      ActionRemember(actions, a, b, FILL_B);
      BucketFill(b);
}

void EmptyA(List* actions, Bucket* a, Bucket* b)
{
      ActionRemember(actions, a, b, EMPTY_A);
      BucketEmpty(a);
}

void EmptyB(List* actions, Bucket* a, Bucket* b)
{
      ActionRemember(actions, a, b, EMPTY_B);
      BucketEmpty(b);
}
```