CS302 -- Project 3 -- Superball!

- CS302 -- Data Structures and Algorithms II
- Spring, 2020
- courtesy of Dr. James S. Plank (http://www.cs.utk.edu/~plank)
- The original file: **http://www.cs.utk.edu/~plank/plank/classes/cs302/Labs/Lab5/** (http://www.cs.utk.edu/~plank/plank/classes/cs302/Labs/Lab5/)

Tue Nov 27 16:37:58 EST 2007. Last revision: Wed Feb 6 11:50:52 EST 2020 (by SJE)

What you hand in

My former UT students are pretty insistent we keep the CS302 Superball! challenge... and Dr. Plank wants me to use all of his labs/notes (if I want), so here it is.

As in the Fall (and previous iterations), this project must be done alone.

You need to submit the source code for two programs: **sb-analyze.cpp** and **sb-play.cpp** as a tar/zip on Canvas (as you have for Challenges).

Also

Every year, someone asks Dr. Plank for the source to **sb-player**. He does not give it out since its "too easy" in his words to solve the lab with it. There are, however, various implementations (see below) and we can ask Dr. Plank to try and make an **sb-player** binary for your machine. Let us know.

There is an **sb-player** binary for macs in **sb-player-mac**.

Plus, in 2015, Alex Teepe wrote a multiplatform Superball player to share. Neither Dr. Plank nor I have not tried it, but please do. Thanks, Alex!

https://drive.google.com/file/d/0B4rzPrfwFCsKbUpwd21pMlgtc1E/view (https://drive.google.com/file/d/0B4rzPrfwFCsKbUpwd21pMlgtc1E/view).

Disjoint Sets

Use the disjoint sets code from the lecture note directory in the course git repo. That means you should include disjoint.h (http://web.eecs.utk.edu/~jplank/plank/classes/cs302/Notes/Disjoint/disjoint.h), and then compile with disjoint-rank.cpp (http://web.eecs.utk.edu/~jplank/plank/classes/cs302/Notes/Disjoint/disjoint-rank.cpp). When you instantiate your disjoint set instance, use "**new DisjointSetByRankWPC**". Since you don't use the other implementations, you don't need to compile with them.

If you don't understand how to compile your program correctly, please ask the TA's or ask on Piazza. DO NOT COPY THE DISJOINT SET CODE AND INCLUDE IT WITH YOUR PROGRAM.

Superball

Superball is a simplistic game that was part of a games CD for Dr. Plank's old Windows 95 box. It works as follows. You have a 8x10 grid which is the game board. Each cell of the game board may be empty or hold a color:

- P - Purple: worth 2 points.
- B - Blue: worth 3 points.
- Y - Yellow: worth 4 points.
- R - Red: worth 5 points.
- G - Green: worth 6 points.

The board starts with five random colors set. On your turn, you may do one of two things:

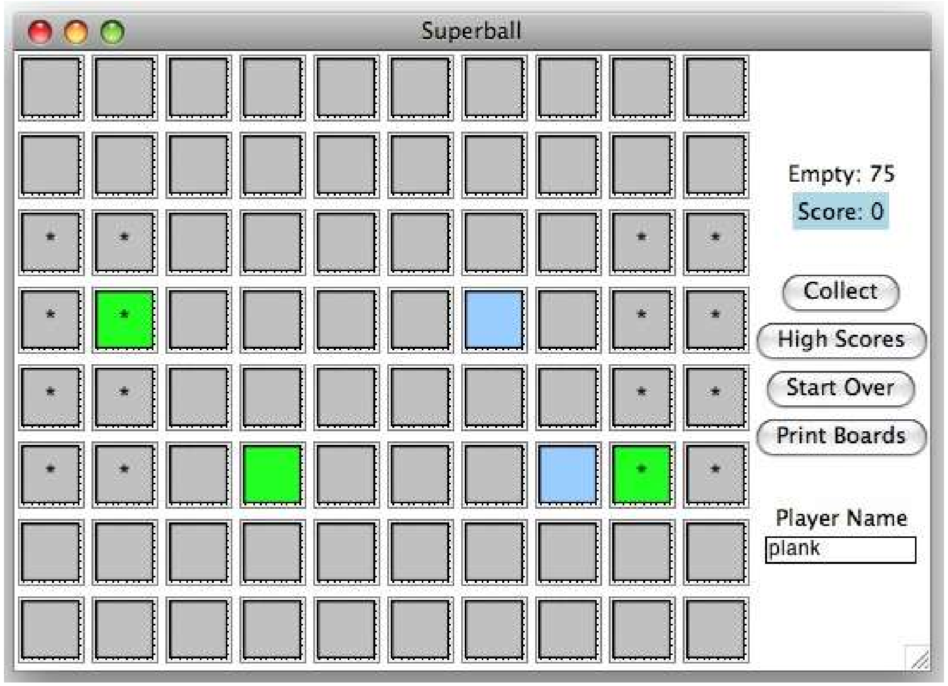
- You may swap two cells. After the swap, five new random cells will be filled with a random colors.
- You may "score" a cell. To score a cell, the cell must be one of the "goal" cells, and there are sixteen of these, in rows 2-5, columns 0, 1, 8 and 9. (Everything is zero indexed). Moreover, there must be at least five touching cells of the same color, one of which must be the goal cell that you want to score. When you score, you get the sum of the cells connected to the cell that you are scoring, and then all of those cells leave the board, and three new random ones are added.

Dr. Plank has a tcl/tk/shell-scripted Superball player at **/home/jplank/Superball**. Simply copy that directory to your home directory:

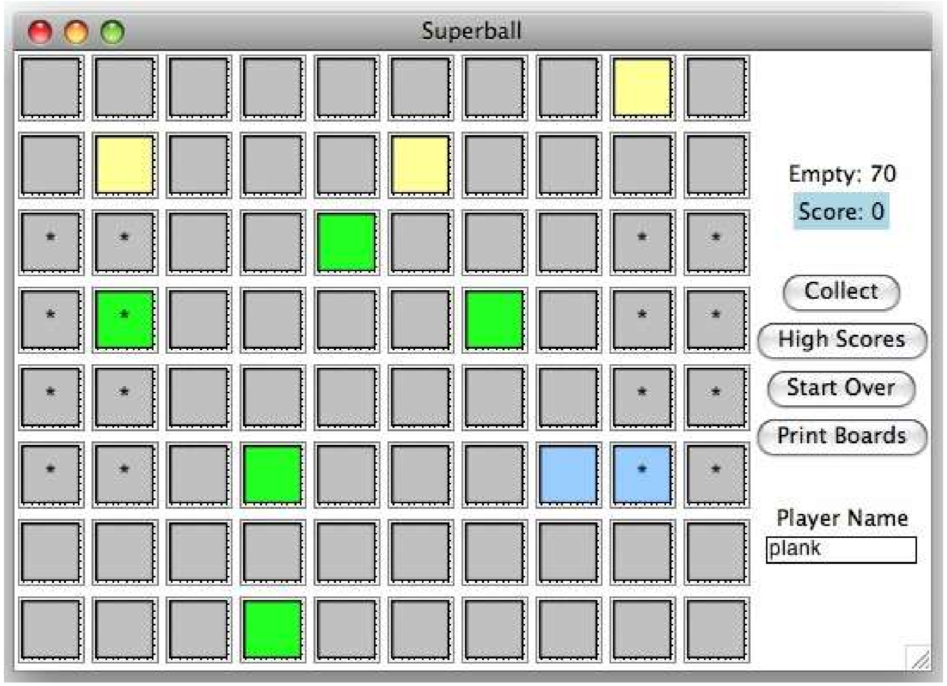
```
UNIX> cp -r /home/jplank/Superball $HOME
```

Then you can play it with **~/Superball/Superball**. The high score probably won't work -- you'll have to change the **open** command in the file **hscore** to the name of your web browser.

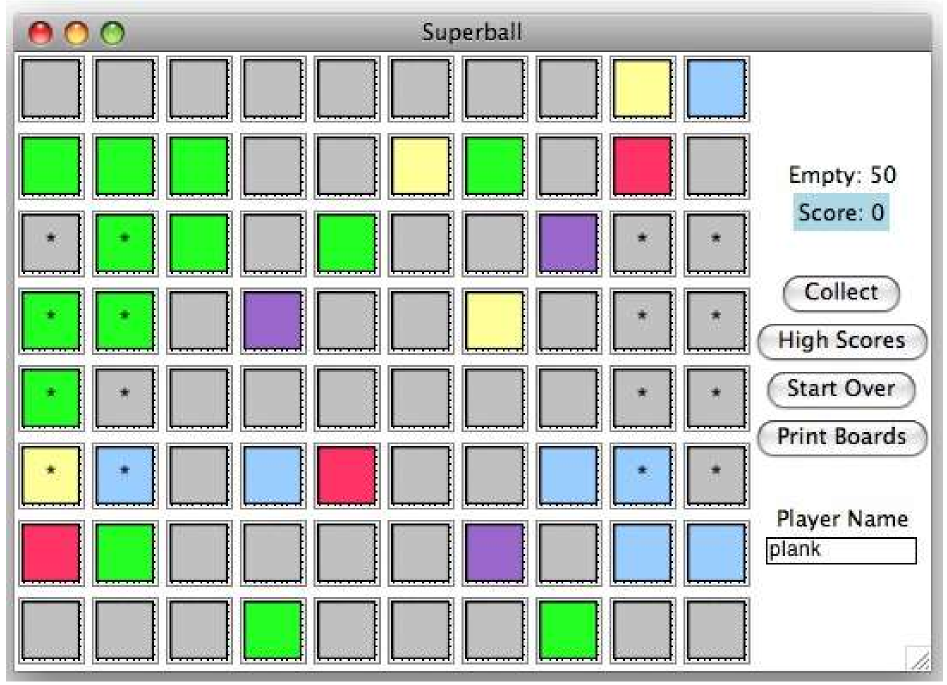
Let's look at some screen shots. Suppose we fire up **Superball**:



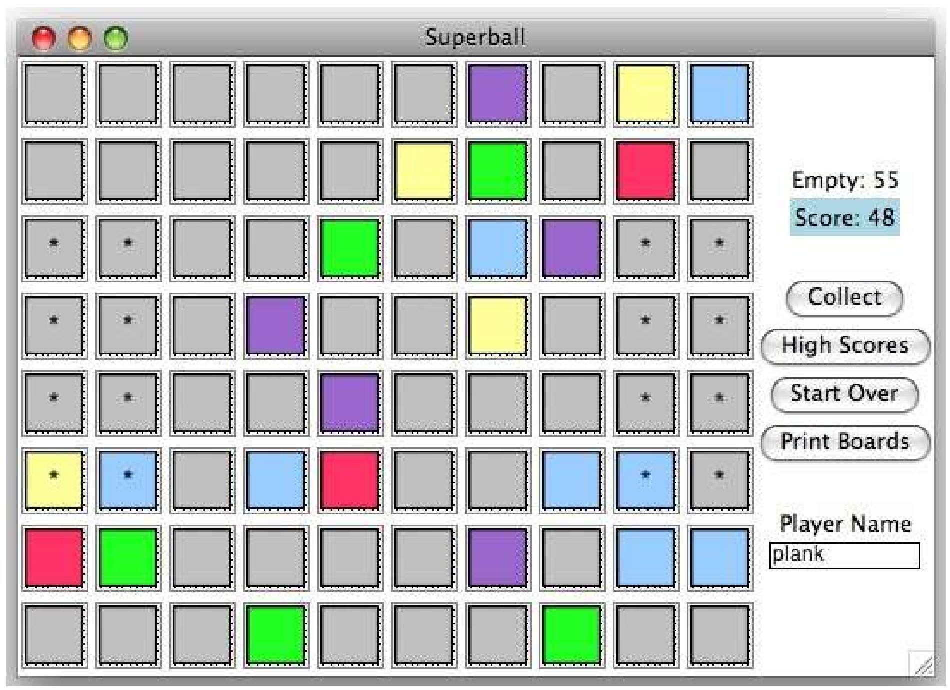
The "goal" cells are marked with asterisks, and there are five non-empty cells. Our only legal action is to swap two cells -- Dr. Plank swaps cells [3,6] and [5,8] below. This will make those two blue cells contiguous. In the game, we can do that by clicking on the two cells that we want to swap. Afterwards, five new cells are put on the screen. Here's the screen shot:



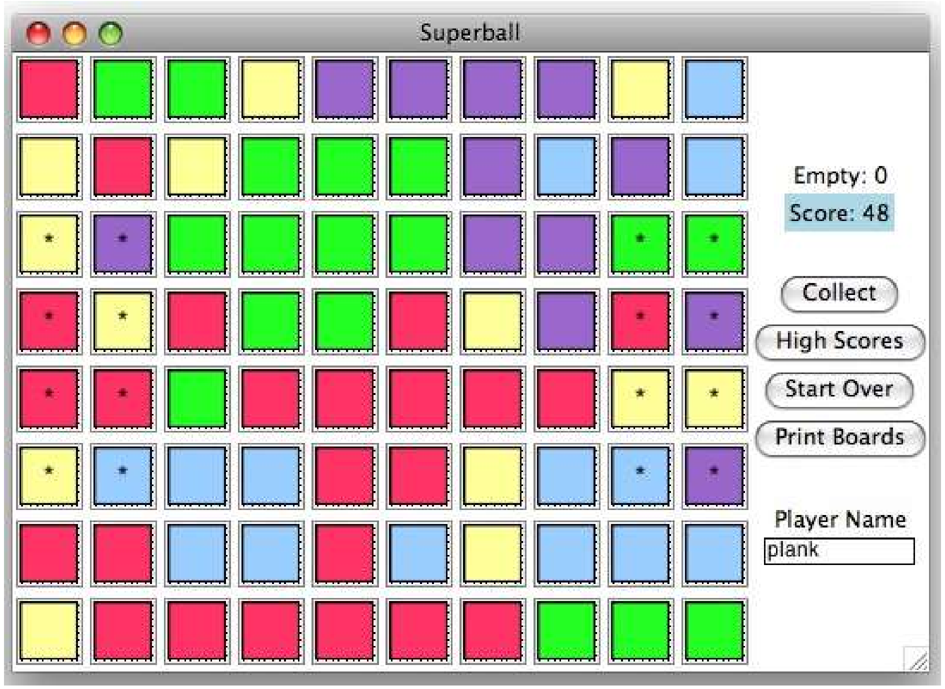
Dr. Plank does a bunch more swaps and ends up with the following board:



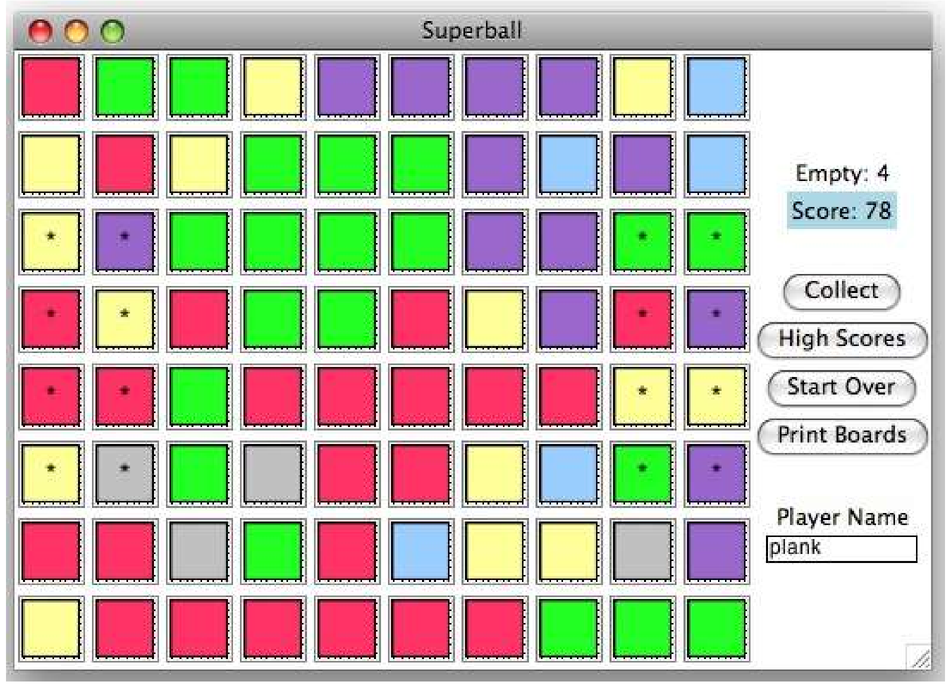
We can score the green cells by clicking on cell [2,1], [3,0], [3,1] or [4,0] and then clicking "Collect". This will score that group of eight green squares, which gets us 48 points (8*6), and three new cells will be added:



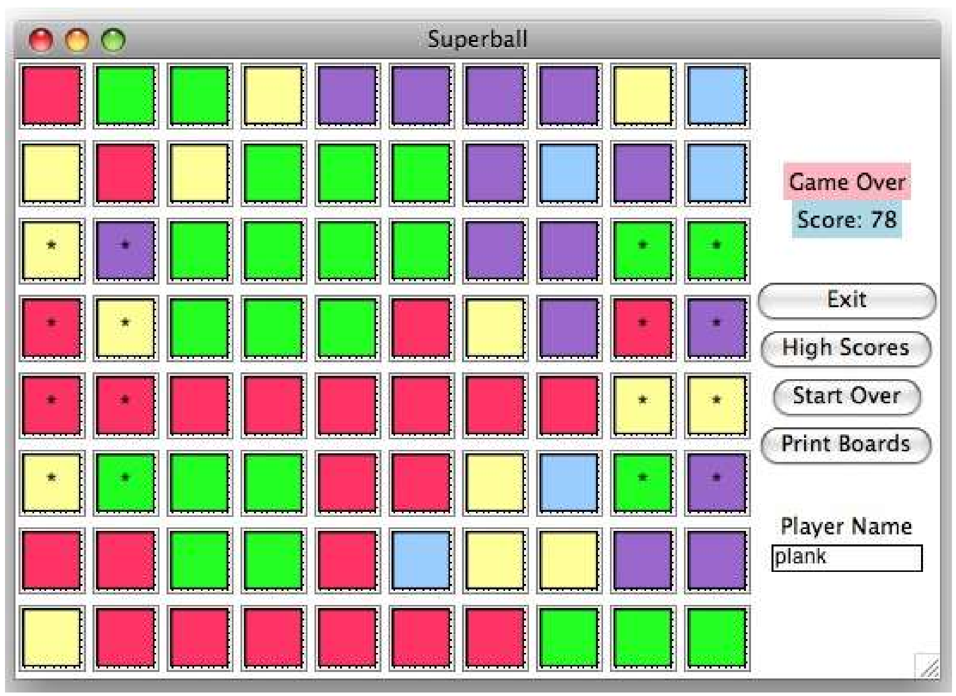
There are no cells to score here (the blues ones in the lower right-hand part of the board only compose a group of four). So Dr. Plank now reverts to swapping. Suppose we keep doing this until we reach:



We're in trouble. Dr. Plank has now got these beautiful groups of red, green and purple cells, but he can't score any of them because they are not connected to a goal. Dang. We can only score those two groups of blue cells. When Dr. Plank does that, he is only left with four open squares, and we can't score anything:



Perhaps Dr. Plank should have been a little more thoughtful while playing the game. Regardless, he is stuck. We simply swap two random squares and end the game:



Oh well -- should have done that swap a little sooner....

For this project, we are going to deal with a text-based version of the game. Our programs will have the following parameters:

- **rows** - the number of rows on the game board. Although the tcl/tk version has that set to eight, our programs will handle any number.
- **cols** - the number of columns on the game board.
- **min-score-size** - the number of contiguous cells that have to be together in order to score them. This is 5 in the tcl/tk version
- **colors** - this must be a string of distinct lower-case letters. They represent that the colors that a cell can have. The point value of the first of these is 2, and each succeeding character is worth one more point. To have the same values as the tcl/tk game, this parameter should be "pbyrg".

Dr. Plank has written an interactive game player. Call it as shown below:

```
UNIX> cd /home/jplank/cs302/Labs/Lab5/
UNIX> sb-player
usage: sb-player rows cols min-score-size colors player interactive(y|n) output(y|n) seed
UNIX> sb-player 8 10 5 pbyrg - y y -
Empty Cells: 75     Score: 0

..........
..........
**b....b**
**....b.**
**.g....**
**......**
..........
..g.......

Your Move:
```

The format of the board is as follows: When a letter is capitalized, it is on a goal cell. Dots and asterisks stand for empty cells -- asterisks are on the goal cells. If you click on the **Print Boards** button in the tcl/tk game, it will print out each board on standard output in that format. That's nice for testing.
You can type two commands:

```
SWAP r1 c1 r2 c2
SCORE r c
```

In the board above, you can't score anything, so you'll have to swap. We'll swap the blue cell in [2,2] with the green one in [7,2]:

```
Your Move: SWAP 2 2 7 2

Empty Cells: 70     Score: 0

.r........
..........
**g....b**
**....b.**
**.g....*Y
**......*P
.....rr...
..b.......

Your Move:
```

It's incredibly tedious -- play along with us:

```
Empty Cells: 70     Score: 0

.r........
..........
**g....b**
**....b.**
**.g....*Y
**......*P
.....rr...
..b.......

Your Move: SWAP 0 1 7 2
```

```
Empty Cells: 65     Score: 0

.b........
..........
**g....bB*
**....b.**
P*.g....RY
**......*P
.....rr...
.gry......

Your Move: SWAP 7 3 4 8
```

```
Empty Cells: 60     Score: 0

.b.......p
....g.....
**g.p..bB*
**r...b.*R
P*.g....YY
**......*P
.....rr...
.grr......

Your Move: SWAP 3 2 7 1
```

```
Empty Cells: 55     Score: 0

.b..r...pp
....g...b.
**g.p..bB*
**g...b.*R
P*.g....YY
**.g....*P
.....rr...
rrrr......

Your Move: SWAP 3 9 0 1
```

```
Empty Cells: 50     Score: 0

.r..rgy.pp
....g...b.
**g.p..bB*
**g...b.*B
P*.g....YY
**.g....*P
p...rrr...
rrrr.p....

Your Move: SWAP 6 0 0 1
```

```
Empty Cells: 45     Score: 0

.p..rgy.pp
.g..g...b.
**g.p..bB*
**g...b.*B
P*.g..y.YY
**.g..yp*P
r...rrr...
rrrr.py...

Your Move: SWAP 5 9 7 6
```

```
Empty Cells: 40     Score: 0

.p..rgy.pp
.g..g...b.
**g.p.pbB*
R*g...by*B
P*.g..y.YY
P*.g..yp*Y
r...rrrb..
rrrr.pp...

Your Move: SWAP 5 0 0 4
```

```
Empty Cells: 35     Score: 0

.p..pgy.pp
.g..g.r.b.
G*g.p.pbB*
R*g...by*B
P*.g..y.YY
R*.g..yp*Y
r..grrrb..
rrrrbppy..

Your Move: SWAP 7 4 1 6
```

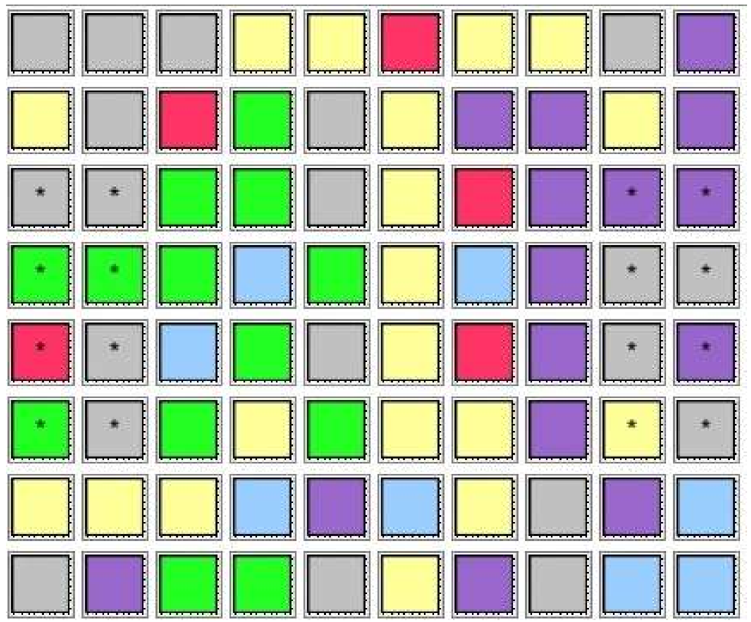
```
Empty Cells: 30     Score: 0

.p..pgy.pp
.g.pg.b.b.
G*g.p.pbB*
R*g.r.by*B
P*pg..y.YY
R*.g.bypBY
r..grrrb..
rrrrrppy..

Your Move: SCORE 5 0
```

```
Empty Cells: 37     Score: 50

.p..pgy.pp
.g.pg.b.by
G*g.p.pbB*
R*g.r.byGB
P*pg..y.YY
**.g.bypBY
...g...b..
.p...ppy..

Your Move:
```

You'll note, we could have scored cell [5,0] when there were 35 empty cells, but Dr. Plank really wanted to make that patch of red cells bigger.

Program #1: Sb-read

Dr. Plank has provided **sb-read.cpp (http://www.cs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/sb-read.cpp)** for us. This program takes the four parameters detailed above, reads in a game board with those parameters and prints out some very basic information. For example, the following board:



May be represented by the following text (in **input-1.txt (http://www.cs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/input-1.txt)**):

```
...yyryy.p
y.rg.yppyp
**gg.yrpPP
GGgbgybp**
R*bg.yrp*P
G*gygyypY*
yyybpby.pb
.pgg.yp.bb
```

When we run **sb-read** on it, we get the following:

```
UNIX> sb-read 8 10 5 pbyrg < input-1.txt
Empty cells:                    20
Non-Empty cells:                60
Number of pieces in goal cells:  8
Sum of their values:            33
UNIX>
```

There are three purple pieces in goal cells, one yellow, three green and one red. That makes a total of 3*2 + 4 + 5 + 3*6 = 33.

You should take a look at **sb-read.cpp (http://www.cs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/sb-read.cpp)**. In particular, look at the **Superball** class:

```
class Superball {
  public:
    Superball(int argc, char **argv);
    int r;
    int c;
    int mss;
    int empty;
    vector <int> board;
    vector <int> goals;
    vector <int> colors;
};
```

Mss is min-score-size. **Empty** is the number of empty cells in the board. **Board** is a vector of **r * c** integers. The element in [i,j] is in entry **board[i*c+j]**, and is either '.', '*' or a lower case letter. **goals** is another array of **r * c** integers. It is equal to 0 if the cell is not a goal cell, and 1 if it is a goal cell. **Colors** is an array of 256 elements, which should be indexed by a letter. Its value is the value of the letter (e.g. in the above example, **colors['p'] = 2**).

sb-read does all manner of error checking for you. It is a nice program from which to build your other programs.

Program #2: Sb-analyze

You are to write this one.

With **sb-analyze**, you are to start with **sb-read.cpp** as a base, and augment it so that it prints out all possible scoring sets. For example, in the above game board (represented by **input-1.txt (http://www.cs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/input-1.txt)**), there are two scoring sets -- the set of 10 purple cells in the upper right-hand corner, and the set of 6 green cells on the left side of the screen. Here is the output to **sb_analyze**:

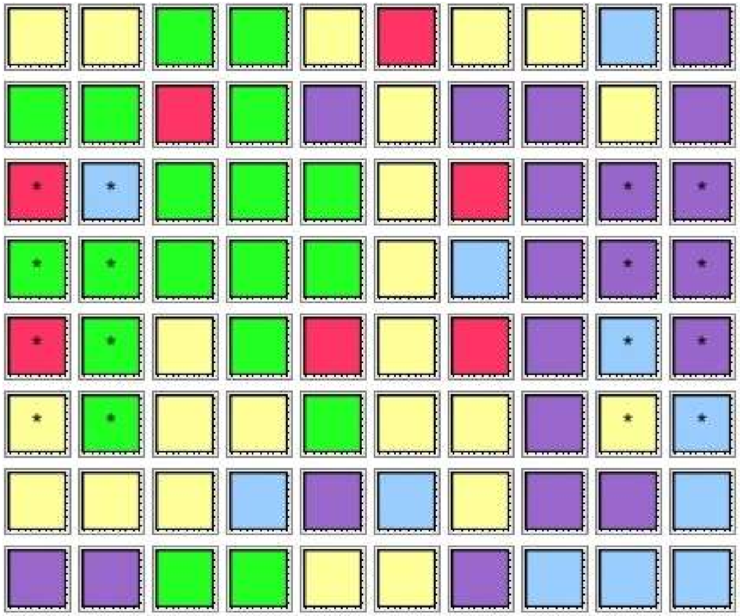
```
UNIX> sb-analyze
usage: sb-analyze rows cols min-score-size colors
UNIX> sb-analyze 8 10 5 pbyrg < input-1.txt
Scoring sets:
  Size: 10  Char: p  Scoring Cell: 2,8
  Size:  6  Char: g  Scoring Cell: 3,0
UNIX>
```

Each set must be printed exactly once, but in any order, and with any legal goal cell. Thus, the following output would also be ok:

```
UNIX> sb-analyze 8 10 5 pbyrg < input-1.txt
Scoring sets:
  Size:  6  Char: g  Scoring Cell: 3,1
  Size: 10  Char: p  Scoring Cell: 2,9
UNIX>
```

Think about how you would use the disjoint sets data structure to implement this -- it is a straightforward connected components application. We would recommend augmenting your **Superball** class with a **DisjointSet**, and then having a method called **analyze_superball()** that performs the analysis.

Here's another example:



This is in the file **input-2.txt (http://www.cs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/input-2.txt)**:

```
yyggyryybp
ggrgpyppyp
RBgggyrpPP
GGgggybpPP
RGygryrpBP
YGyygyypYB
yyybpbyppb
ppggyypbbb
```

```
UNIX> sb-analyze 8 10 5 pbyrg < input-2.txt
Scoring sets:
  Size: 14  Char: g  Scoring Cell: 5,1
  Size: 15  Char: p  Scoring Cell: 4,9
  Size:  7  Char: y  Scoring Cell: 5,0
  Size:  5  Char: b  Scoring Cell: 5,9
UNIX>
```

Program #3: Sb-play

Your next program takes the same arguments and input as **sb-analyze**. However, now its job is to print a single move as would be accepted as input for the **sb-player** program. In other words, it needs to output a SWAP or SCORE line with legal values.

If you have fewer than five pieces and cannot score any, you will lose the game -- you should do that by swapping two legal pieces so that the game can end.

The **sb-player** program takes as its 5th argument the name of a program that it will use for input. Dr. Plank also also provided three programs - **sb-play**, **sb-play2** and **sb-play3** in that directory. **sb-play** simply swaps two random cells until there are fewer than five empty, then it scores a set if it can. The other two are smarter, but are by no means the best one can do.

Here's **sb-player** running on **sb-play2** (note, **sb-player** creates a temporary file, so you must run it from your own directory):

```
UNIX> /home/jplank/cs302/Labs/Lab5/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/sb-play2 y y -
Empty Cells: 75     Score: 0

g.........
..........
**......**
*Pr.....**
**......**
**..p...**
........b.
..........

Type Return for the next play
```

It waits for you to press the return key. When you do so, it will send the game board to **/home/jplank/cs302/Labs/Lab5/sp-play2** and perform the output. Here's what happens:

```
Move is: SWAP 5 4 3 2

Empty Cells: 70     Score: 0

g........g
.......y..
**......**
*Pp.....**
**......G*
**..r...**
..g.....b.
........g.

Type Return for the next play
```

You can bet that the next move will swap that **b** with one of the **g**'s:

```
Move is: SWAP 6 8 0 0

Empty Cells: 65     Score: 0

b........g
.......y..
**..b...**
*Pp.g...**
**.....gG*
**..r...**
..g.....g.
.p...p..g.

Type Return for the next play
```

And so on. If you run it with **n** for the 6th argument, it will simply run the program without your input:

```
UNIX> /home/jplank/cs302/Labs/Lab5/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/sb-play2 n y -
Empty Cells: 75     Score: 0

..........
..........
**......**
**y..y..**
**......**
*P......**
..........
......p.g.

Move is: SWAP 3 5 3 2
```

... a bunch of output skipped...

```
Empty Cells:  1     Score: 505

yyrrgggpyy
grrbppg.yg
GYbgygggPB
GBggpgbpPB
PPgggggrYB
YBbybgpbYR
pprrrggggr
byyrppppgg

Move is: SWAP 0 1 7 5

Game over.  Final score = 505
UNIX>
```

Even though there were no good moves at the end, the program did a final SWAP so that the game could finish.
If you run with the 7th argument as **n**, it will only print out the end result, and the last argument can specify a seed (it uses the current time if that argument is "-"), so that you can compare multiple players on the same game:

```
UNIX> /home/jplank/cs302/Labs/Lab5/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/sb-play n n 1
Game over.  Final score = 0
UNIX> /home/jplank/cs302/Labs/Lab5/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/sb-play2 n n 1
Game over.  Final score = 855
UNIX> /home/jplank/cs302/Labs/Lab5/sb-player 8 10 5 pbyrg /home/jplank/cs302/Labs/Lab5/sb-play3 n n 1
Game over.  Final score = 2572
UNIX>
```

It can take a while for these to run -- if it appears to be hanging, send the process a **QUIT** signal and it will print out what the current score is.

Shell Script to Run Multiple Times

The file **run_multiple.sh (http://www.cs.utk.edu/~jplank/plank/classes/cs302/Labs/Lab5/run_multiple.sh)** is a shell script to run the player on multiple seeds and average the results. Examples:

```
UNIX> sh run_multiple.sh
usage: sh run_multiple.sh r c mss colors player nruns starting_seed
UNIX> sh run_multiple.sh 8 10 5 pbyrg sb-play 10 1
Run   1 - Score:     38  - Average     38.000
Run   2 - Score:      0  - Average     19.000
Run   3 - Score:      0  - Average     12.667
Run   4 - Score:     57  - Average     23.750
Run   5 - Score:      0  - Average     19.000
Run   6 - Score:      0  - Average     15.833
Run   7 - Score:     89  - Average     26.286
Run   8 - Score:     15  - Average     24.875
Run   9 - Score:      0  - Average     22.111
Run  10 - Score:     20  - Average     21.900
UNIX> sh run_multiple.sh 8 10 5 pbyrg sb-play2 10 1
Run   1 - Score:    855  - Average    855.000
Run   2 - Score:    979  - Average    917.000
Run   3 - Score:    650  - Average    828.000
Run   4 - Score:    833  - Average    829.250
Run   5 - Score:    832  - Average    829.800
Run   6 - Score:   3326  - Average   1245.833
Run   7 - Score:   1507  - Average   1283.143
Run   8 - Score:   3643  - Average   1578.125
Run   9 - Score:    610  - Average   1470.556
Run  10 - Score:    862  - Average   1409.700
UNIX> sh run_multiple.sh 8 10 5 pbyrg sb-play3 10 1
Run   1 - Score:   2572  - Average   2572.000
Run   2 - Score:   2708  - Average   2640.000
Run   3 - Score:    745  - Average   2008.333
Run   4 - Score:    424  - Average   1612.250
Run   5 - Score:   1888  - Average   1667.400
Run   6 - Score:   7140  - Average   2579.500
Run   7 - Score:   3475  - Average   2707.429
Run   8 - Score:   1701  - Average   2581.625
Run   9 - Score:   2699  - Average   2594.667
Run  10 - Score:   2291  - Average   2564.300
UNIX>
```

Obviously, to get a meaningful average, many more runs (than 10) will be required.

Oh, and make your programs run in reasonable time. Roughly 5 seconds for every thousand points, and if you are burning all that time, your program better be killing Dr. Plank's....

The Superball Challenge

To get credit, your player needs to average over 100 points on runs of 100 games.
We will run a Superball tournament with all of your players with extra lab points going to the winners:

- 1st place: 20 extra lab points.
- 2nd place: 10 extra lab points.
- 3rd place: 5 extra lab points.

Dr. Plank and I have previously performed the challenge eight times:

- CS140 in 2007.
- CS302 in 2010.
- CS302 in 2011.
- CS302 in 2012.
- CS302 in 2013.
- CS302 in 2014.
- CS302 in 2015.
- CS302 in 2018.
- CS302 in 2019 (SJE)

Here's the Superball Challenge Hall Of Fame (scores over 650):

Rank	Average	Name	Semester
1	31814.13	Grant Bruer	CS302, Fall, 2015
2	24278.49	Alexander Teepe	CS302, Fall, 2015
3	17367.77	Joseph Connor	CS302, Fall, 2014
4	17021.37	Cory Walker	CS302, Fall, 2014
5	16963.40	Seth Kitchens	CS302, Fall, 2015
6	14555.83	Ben Arnold (Tie)	CS302, Fall, 2012
7	14555.83	Adam Disney (Tie)	CS302, Fall, 2011
8	13657.79	Isaak Sikkema	CS302, Fall, 2018
9	12963.47	Jake Davis	CS302, Fall, 2014
10	12634.29	Jake Lamberson	CS302, Fall, 2014
11	11722.05	Parker Mitchell	CS302, Fall, 2014
12	11418.77	James Pickens	CS302, Fall, 2014
13	11380.74	Nathan Ziebart	CS302, Fall, 2011
14	11291.39	Michael Jugan	CS302, Fall, 2010
15	10576.96	Tyler Shields	CS302, Fall, 2014
16	8770.67	Nathan Swartz	CS302, Spring, 2019
17	7475.07	Jared Smith	CS302, Fall, 2014

18	7216.28	Michael Bowie	CS302, Fall, 2018
19	7003.56	Andrew LaPrise	CS302, Fall, 2011
20	6100.28	Chris Nagy	CS302, Fall, 2015
21	5467.56	Tyler Marshall	CS302, Fall, 2013
22	5262.80	Harry Channing	CS302, Fall, 2018
23	5116.13	Kyle Bashour	CS302, Fall, 2014
24	4808.03	Matt Baumgartner	CS302, Fall, 2010
25	4586.51	Jeramy Harrison	CS302, Fall, 2013
26	4531.96	Philip Hicks	CS302, Spring, 2019
27	4057.08	Phillip McKnight	CS302, Fall, 2015
28	3882.53	Pranshu Bansal	CS302, Fall, 2013
29	3882.28	Kemal Fidan	CS302, Fall, 2018
30	3852.87	Yaohung Tsai	CS302, Fall, 2015
31	3849.24	Chris Richardson	CS302, Fall, 2010
32	3809.41	Arthur Vidineyev	CS302, Fall, 2015
33	3588.35	Kevin Dunn	CS302, Fall, 2014
34	3464.83	Patrick Slavick	CS302, Fall, 2012
35	3436.21	sb-play3	CS140, Fall, 2007
36	3400.50	Kody Bloodworth	CS302, Fall, 2018
37	3080.15	Andrew Messing	CS302, Fall, 2013
38	2903.38	Adam LaClair	CS302, Fall, 2013
39	2555.36	Mohammad Fathi	CS302, Fall, 2014
40	2532.89	Trevor Sharpe	CS302, Fall, 2015
41	2521.44	Justus Camp	CS302, Fall, 2018
42	2335.88	Mark Clark	CS302, Fall, 2012
43	2307.16	John Burnum	CS302, Fall, 2012
44	2205.17	Shawn Cox	CS302, Fall, 2011
45	2163.70	Alex Wetherington	CS302, Fall, 2011
46	2134.99	Julian Kohann	CS302, Fall, 2013
47	2011.38	Wells Phillip	CS302, Fall, 2015
48	1919.72	Ravi Patel	CS302, spring, 2019
49	1778.83	Keith Clinart	CS302, Fall, 2011
50	1740.19	Luke Bechtel	CS302, Fall, 2014
51	1634.49	William Brummette	CS302, Fall, 2013
52	1602.83	Forrest Sable	CS302, Fall, 2014
53	1470.84	Christopher Tester	CS302, Fall, 2014
54	1433.48	Xiao Zhou	CS302, Fall, 2015
55	1430.54	Jonathan Burns	CS302, Fall, 2018
56	1340.32	John Murray	CS302, Fall, 2012
57	1329.34	Benjamin Brock	CS302, Fall, 2013
58	1257.56	Dylan Lee	CS302, Fall, 2018

59	1202.06	Bandara	CS302, Fall, 2014
60	1149.80	Will Houston	CS302, Fall, 2010
61	1119.85	Kevin Chiang	CS302, Fall, 2014
62	1096.48	Daniel Cash	CS302, Fall, 2011
63	1059.91	Kaleb McClure	CS302, Fall, 2013
64	1058.26	sb-play2	CS140, Fall, 2007
65	1029.63	Lydia San George	CS302, Fall, 2018
66	972.36	Erik Rutledge	CS302, Fall, 2013
67	959.79	Daniel Nichols	CS302, Fall, 2018
68	917.92	Vasu Kalaria	CS302, Fall, 2015
69	908.09	Chris Rains	CS302, Fall, 2012
70	875.44	Allen McBride	CS302, Fall, 2012
71	840.94	Spencer Howell	CS302, Fall, 2018
72	830.79	David Cunningham	CS302, Fall, 2014
73	810.17	Collin Bell	CS302, Fall, 2012
74	763.58	Jacob Lambert	CS302, Fall, 2013
75	703.67	Scott Marcus	CS302, Fall, 2015
76	703.00	Don Lopez	CS140, Fall, 2007
77	700.90	Tony Abston	CS302, Fall, 2015
78	682.56	Jackson Collier	CS302, Fall, 2014
79	677.83	KC Bentjen	CS302, Fall, 2011
80	665.60	Joshua Clark	CS302, Fall, 2012
81	659.96	Warren Dewit	CS302, Fall, 2010
82	654.67	Coburn Brandon	CS302, Fall, 2015
83	650.98	Joaquin Bujalance	CS140, Fall, 2007

Hints

Play the game for a bit to try to figure out some strategies. However, one good way to write a game player is to figure out a way to come up with a rating for a game board. Then when you are faced with making a move, you analyze all potential moves by trying them out and choosing the one that gives you the resulting board with the highest rating.