The Application of Backtracking and Greedy Algorithm to Jawbreaker

Martha Monica (13510080)
Informatics Engineering
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganeca Street 10 Bandung 40132, Indonesia
martha.monica@students.itb.ac.id

***Abstract*— Jawbreaker is a simple ball game which is played by removing the ball in the board as much as possible. The more adjacent pieces removed, the more point will the players got. This paper will compare the application of backtracking and greedy algorithm in playing Jawbreaker to find the way to achieve the best high score.**

***Index Terms*—Jawbreaker, backtracking, greedy, algorithm.**

I. INTRODUCTION

Everybody always wants to simplify or optimize the solution of their problem. This reason encourage many people innovate many ways to simplify or optimize it. One of those is the invention of many algorithms in solving problem. The application of the algorithms is not only in mathematical problems, but also can be used to solve some problem people face commonly like how to find the fastest way to reach their home, how to ship their goods in an effective way, even to find the way to solve some game.

This paper will compare about the application of backtracking and greedy algorithm in playing one of the most popular game, Jawbreaker. Jawbreaker is a simple game which is played just by removing the ball in the board as much as possible. The more balls removed, the more points will the player gets.

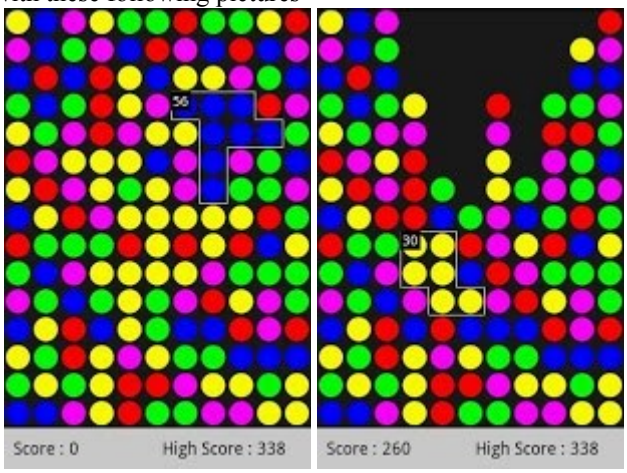
Backtracking algorithm is an algorithm which is based on DFS algorithm for finding all solutions to some computational problem in. This algorithm is the improvement of brute force algorithm where we don't have to check all possible solution, only the steps which are forwarding to the solution will be processed. Because of the improvement, the execution time will lessen. The implementation of this algorithm is a typical form of recursive algorithm.

Greedy algorithm is one popular method using in optimization problems. This algorithm is so simple and straightforward where naturally the decision which will be chosen in one time is the step which will give the best result that time. In this algorithm, we didn't care whether the decision is the best overall solution of the problem, we just taking the best solution for every condition.

II. JAWBREAKER

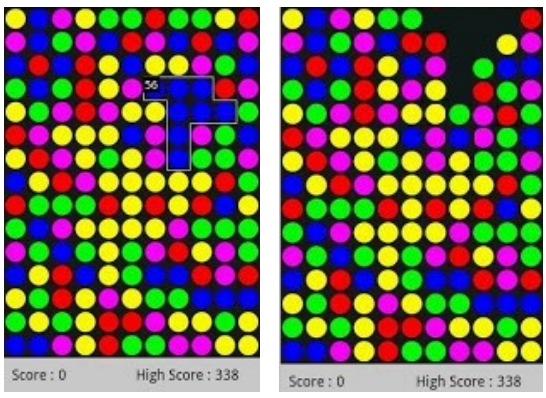
Jawbreaker is a simple game which is played just by removing the ball in the board as much as possible. The more balls removed, the more points will the player gets.

The game play of the Jawbreaker game can be display with these following pictures



The goal of this game is to break down/remove the balls with same color (minimal 3 balls) and achieve the highest score when there are no moves left to break down the ball. When player picks to break down one color, the balls on the upper side will drop down until there are no spaces left in the bottom side.

The following pictures is the visualization of the movement of the balls



Before After

The score is based on the number of adjacent balls with the same color with the formula SCORE = N * (N-1) where N is the number of adjacent balls with the same color.

That's why strategy is needed to choose the right step so the players can form more adjacent balls with the same color to get the higher score.

III. ALGORITHM

3.1 Greedy Algorithm

In common, greedy algorithm scheme can be define as following steps

Initialize subset S with empty value

i. Choose a candidate with selection function from candidate set C
ii. Remove C with the candidate chosen before
iii. Check whether the candidate will give a feasible result (check it using feasibility function) or not. If it gives feasible result, put the candidate to the result set, but if it is not, remove the candidate and never consider to process the candidate again
iv. Check whether the result set has given the complete result (check it with solution function) or not. If it is the complete result, stop the process, but if it is not the result, replay again from the step (ii).

The pseudo code of Greedy algorithm can be written as following

```
function SELECT (input c:candidate_set) → candidate
{ return a candidate from c based upon some criteria}

function SOLUTION (input S:candidate_set) → boolean
{ check whether S is the complete solution or not }

function FEASIBLE (input S:candidate_set) → boolean
{ check whether S is the feasible solution or not }

function greedy (input c: candidate_set ) → candidate_set
{ return the solution of the optimation problems with greedy algorithm
Input : candidate set C
Output : solution set which type is candidate set
}

DECLARATION
    x : candidate
    S : candidate_set

ALGORITHM
    S ← ( ) { initialize S with empty value}
    while not (SOLUTION (S) ) and (c ≠ ( ) ) do
        x ← SELECT (C) {choose a candidate from C}
        c ← c – {x} { remove x from c }
        if FEASIBLE ( S ∪ {x}} then
            S ← S ∪ {x}
        endif
        endwhile
{SOLUTION (S) or C = { }}
if SOLUTION(S) then
    → S
else
    output('Solution doesn't found')
endif
```

3.2 Backtracking algorithm

Generally, the implementation of backtracking algorithm can be define as following steps

i. Solution is search by making a path from the root to the leaves. The path formation using DFS method (depth searching). The live node will be spread and named as Expand node.
ii. Every time the live node expands, the length of the path will be increased. If it forwarding to the result, the live node E will not be "killed" (dead node). The dead node will not be expanded.
iii. If the path creation ended in a dead node, the process will be continued by expanding the other child node. If there are no child node remains, we will backtrack to the nearest parent node. This node will be the next live node. Then, new path will be formed until the solution is found.
iv. The process will stop if we have reached the complete solution (path from the root to the leaves) or there is no more live node to be trackbacked.

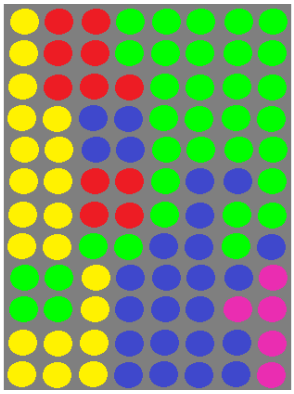
The pseudo code of backtrack algorithm can be written as following

```
procedure printsolution (input x : TabelInt)
{ print the solution
  Input : x[1],x[2], .. ,x[n]
  Output : print the value of x[1], x[2], … , x[n]}

DECLARATION
    k : integer

ALGORITHM
    for k←1 to n do
        output(x[k])
    endfor

procedure backtrack(input R: integer)
{ search all solution with backtracking method with recursive scheme
Input : k, which is the index in the solution vector component, x[k]
Output : Solution x = (x[1], x[2], … , x[n])}

ALGORITHM
foreach x[k] which hasn't been checked before where (x[k] ← T[k] ) and B(x[1],x[2],…, x[k]) = true do
    if (x[1],x[2], …, x[k] ) is path from root to leaves then
        printsolution (x)
    endif
    backtrack(k+1)
endfor
```

IV. Discussion and Analysis

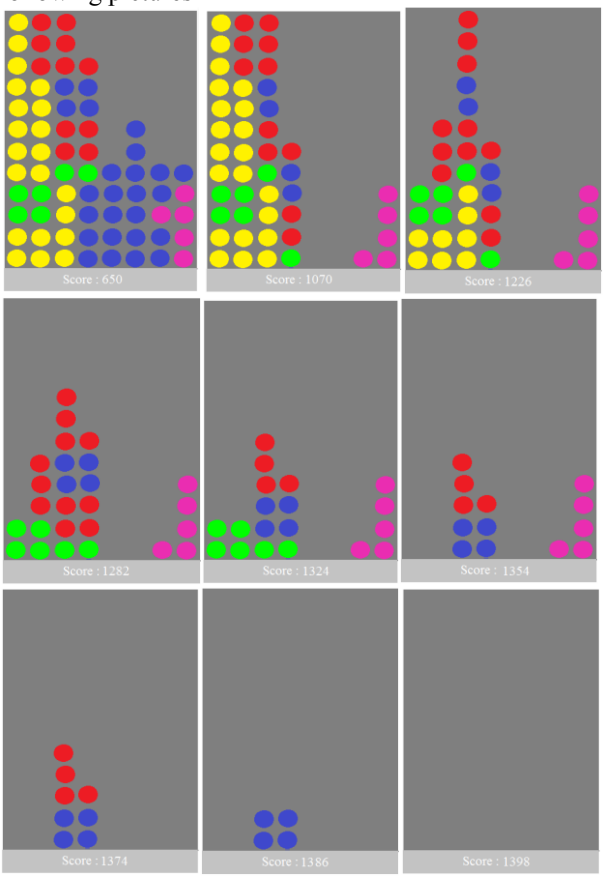
4.1 Greedy Algorithm

The implementation of the greedy algorithm to the Jawbreaker game is to find the biggest number of adjacent balls with same colors. For the example, we can visualize the movement will be chosen when using greedy algorithm for this following case



As the definition before, the greedy algorithm will choose to break down the ball which will give the largest score. In this state, we can find that the largest score will be gotten if we break down the green balls in the right upper side. It will give 26 x (26-1) = 650 to the score.

The following step and score will be visualize in following pictures



When the game is finished, the implementation of greedy algorithm will score 1498.

In the implementation of different color balls can be represented with different integer value. The pseudo code of the implementation greedy algorithm to solve Jawbreaker can be written as following

```
type Coordinate {
    row : integer
    col : integer }

function searchSame(input r,c : integer)→ integer
{ to count and mark the adjacent ball with same color to avoid duplicate checking }

DECLARATION
    temp : integer

ALGORITHM
    temp ← T[r,c]
    B[r,c] ← false
    count ← count + 1
    { spread to all direction}
    if (B[r-1,c]=true) and (T[r-1,c]=temp) then
        searchSame(r-1,c,T)
    endif
    if (B[r+1,c]=true) and (T[r+1,c]=temp) then
        searchSame(r+1,c,T)
    endif
    if (B[r,c-1]=true) and (T[r,c-1]=temp) then
        searchSame(r,c-1,T)
    endif
    if (B[r,c+1]=true) and (T[r,c+1]=temp) then
        searchSame(r,c+1,T)
    endif
    else {all adjacent ball color balls counted and marked}
        → count

function searchMax( ) → Coordinate
{ to find the ball position which will give biggest score, return -1,-1 if no moves}
DECLARATION
    i,j,max : integer
    pos : Coordinate

ALGORITHM
    temp ← 0
    for i=1 to MaxRow do
        for j=1 to MaxCol do
            if (B[i,j]=false) then
                if (searchSame(i,j)>temp) then
                    temp ← searchSame(i,j)
                    pos ← (i,j)
                endif
            endif
        endfor
    endfor
    setScore(temp)
    → pos
```

```
procedure remove (input i,j : integer)
{ will remove selected balls }
DECLARATION
    i,j,temp : integer

ALGORITHM
    temp ← T[r,c]
    T[r,c] ← -1
    { spread to all direction}
    if (T[r-1,c]=temp) then
        remove(r-1,c,T)
    endif
    if (T[r+1,c]=temp) then
        remove(r+1,c,T)
    endif
    if (T[r,c-1]=temp) then
        remove(r,c-1,T)
    endif
    if (T[r,c+1]=temp) then
        remove(r,c+1,T)
    endif

procedure dropdown ()
{ will adjust the position after remove selected balls }
DECLARATION
    i,j,k,max : integer

ALGORITHM
    i ← MaxCol
    while (i>0) do
        j ← MaxRow
        while (j>0) do
            k ← j
            if  (T[j,i] = -1) then
            while (T[k,i] = -1) and (k>=1) do
                k ← k-1
                if T[k,i] ≠ -1 then
                    T[j,i] ← T[k,i]
                    T[k,i] ← -1
                endif
            endwhile
            j ← j-1
        endwhile
        i ← i-1
    endwhile

procedure setScore( input s: integer)
{ to set the score for the hit}
ALGORITHM
    sc ← s x (s-1)


MAIN PROGRAM

DECLARATION
    i,j,score,sc : integer
```

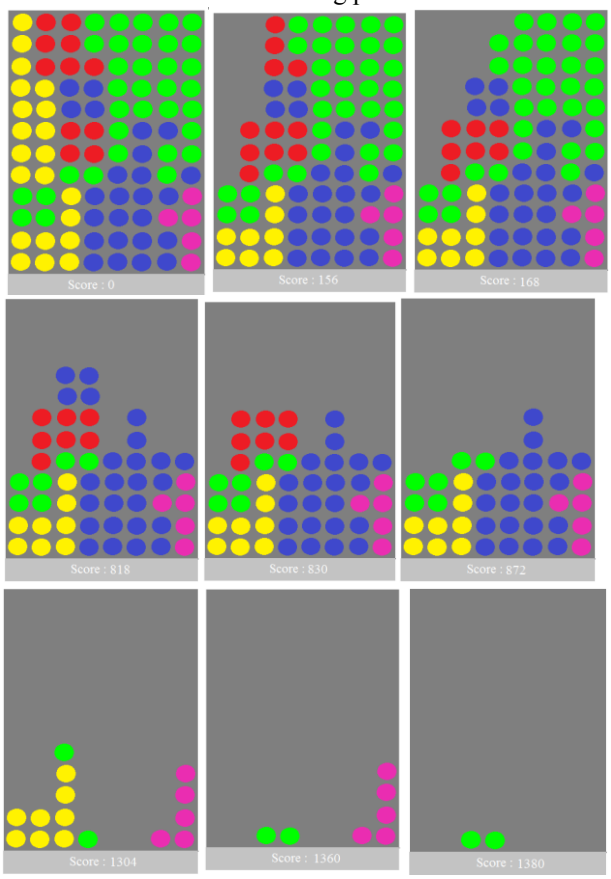
```
ALGORITHM
    input (T)
    while (searchMax() ≠ (-1,-1) do
        score ← score + sc
        remove(pos)
        dropdown()
    endwhile
```

If *we* use this algorithm, we can't ensure that it will achieve the best score for the problem because it only considers the best option in every step. However, the process will be more

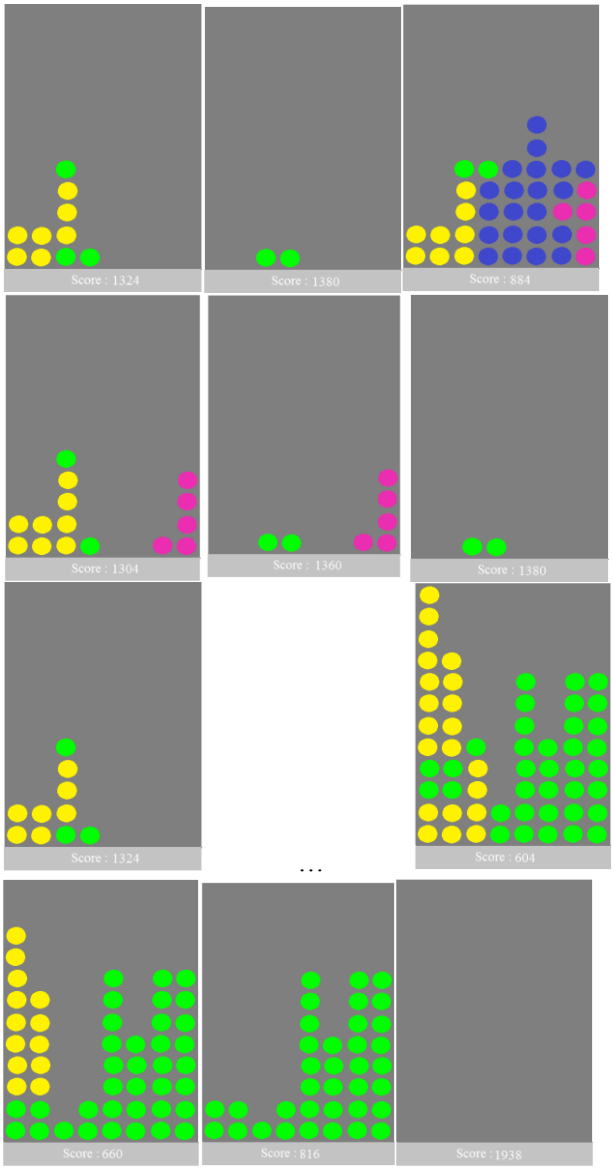
4.2 Backtrack Algorithm

In the backtrack algorithm, every possible step leading to the solution. When there is no more child node to expand, the process will be move to the parent nearest node and expand it as live node. Contrary with the greedy algorithm, if we use backtrack algorithm, we will get the highest score for the problem because we count every possible step. However, the process will take longer time and memory size because there are so much recursive processes used in this algorithm.

With the same case used in greedy algorithm, step and score will be visualize in following pictures



When the process reaches this step, the best score recorded is 1380 and there are no more moves, the process will backtrack and move to other potential solution. It can be visualized with this following image



The process will continue until all possibilities have been checked. In one of the solution, we find a higher score achieved, even more then what we get if we use greedy algorithm.

The pseudo code of the implementation greedy algorithm to solve Jawbreaker can be written as following

```
function searchSame(input r,c : integer)→ integer
{ to count and mark the adjacent ball with same color to
avoid duplicate checking }

DECLARATION
    temp : integer

ALGORITHM
    temp ← T[r,c]
    B[r,c] ← false
    count ← count + 1
    { spread to all direction}
    if (B[r-1,c]=true) and (T[r-1,c]=temp) then
        searchSame(r-1,c,T)
    endif
    if (B[r+1,c]=true) and (T[r+1,c]=temp) then
        searchSame(r+1,c,T)
    endif
    if (B[r,c-1]=true) and (T[r,c-1]=temp) then
        searchSame(r,c-1,T)
    endif
    if (B[r,c+1]=true) and (T[r,c+1]=temp) then
        searchSame(r,c+1,T)
    endif
    else {all adjacent ball color balls counted and marked}
        → count
procedure remove (input i,j : integer)
{ will remove selected balls }
DECLARATION
    i,j,temp : integer
ALGORITHM
    temp ← T[r,c]
    T[r,c] ← -1
    { spread to all direction}
    if (T[r-1,c]=temp) then
        remove(r-1,c,T)
    endif
    if (T[r+1,c]=temp) then
        remove(r+1,c,T)
    endif
    if (T[r,c-1]=temp) then
        remove(r,c-1,T)
    endif
    if (T[r,c+1]=temp) then
        remove(r,c+1,T)
    endif
procedure dropdown ()
{ will adjust the position after remove selected balls }
DECLARATION
    i,j,k,max : integer
ALGORITHM
    i ← MaxCol
    while (i>0) do
        j ← MaxRow
        while (j>0) do
            k ← j
            if (T[j,i] = -1) then
            while (T[k,i] = -1) and (k>=1) do
                k ← k-1
                if T[k,i] ≠ -1 then
                    T[j,i] ← T[k,i]
                    T[k,i] ← -1
                endif
            endwhile
            j ← j-1
        endwhile
        i ← i-1
    endwhile
```

```
function isFinish( ) → boolean
{ to check whether it reach finish or not}
DECLARATION
    i,j : integer
ALGORITHM
    for i=1 to MaxRow do
        for j=1 to MaxCol do
            if (B[i,j]=false) then
                if (searchSame(i,j)>2) then
                    → false
                endif
            endif
        endfor
    endfor
    {no moves left}
    → true

procedure count(input score,input T:TabInt)
{ recursive method to search all solution and pick the biggest score }
DECLARATION
    i,j,temp : integer
    Tc : TabInt

ALGORITHM
  if not isFinish() then
     Tc ← T {backup the table}
     for i=1 to MaxRow do
        for j=1 to MaxCol do
              if (B[i,j]=false) then
                 if (searchSame(i,j)>temp) then
                       temp ← searchSame(i,j)
                       score ←score+(temp x(temp-1))
                       if (score>max) then
                          max ←score
                       remove(i,j)
                       dropdown
                       count (score,T)
                       { backtrack }
                       T ← Tc {put back the value}
                       score ←score-(temp x(temp-1))
                 endif
              endif
        endfor
     endfor
  endif

MAIN PROGRAM

DECLARATION
    max: integer
    T : TabInt
ALGORITHM
    input (T)
    count (0, T)
    output(max)
```

V. CONCLUSION

The implementation of greedy and backtracking algorithm can be used to solve the Jawbreaker game. Both algorithms give different advantages. When we use the greedy algorithm, we can't always get the highest score for the problem, but it will use less time and memory. In the other side, backtrack algorithm will give the best solution for the problem, but it will take more time to compute. If we compare the result from the greedy and backtrack algorithm, we can say that for solving the Jawbreaker game, it is better to use backtrack algorithm, because it will always give the best way to achieve the best score.

REFERENCES

[1] Thomas H. Cormen; Charles E. Leiserson, Ronald R. Rivest, Cliff Stein, Introduction to Algorithms,McGraw-Hill(1990), page 389

[2] J. Bang-Jensen, G. Gutin and A. Yeo, When the greedy algorithm fails. Discrete Optimization 1 (2004), page 121–127

[3] Donald E. Knuth, The Art of Computer Programming. Addison-Wesley (1968)

[4] Gilles Brassard, Paul Bratley (1995). Fundamentals of Algorithmics. Prentice-Hall.

STATEMENT

With this I state that this paper is my own writing, not adaption, or translation from other's paper, and not plagiarism.

Bandung, December 21th 2012

Martha Monica (13510080)