Solving Qwirkle with Backtracking

Luis Mariano Ramírez Segura
Algorithm Analysis
Instituto Tecnológico de Costa Rica
Project Repository: github.com/Lumanter/Qwirkle

***Abstract*— In this work backtracking programming technique is used to implement two Qwirkle solvers (simple and smarter) in Python. After proving it as an effective technique the solvers are compare against each other and the results showed the smarter solver to have a slightly better winning ratio and nearly twice the runtime.**

Keywords*— *programming, backtracking, Qwirkle, solver, Python

I. Introduction

The aim of this project is to implement a Qwirkle solver using the backtracking technique. Qwirkle is a board game in which tiles of 6 different shapes and colors are connected in lines to score points, a linked line can have a maximum of 6 tiles and they all have to share one aspect, be the shape or color. When a line of 6 tiles is formed the player scores a Qwirkle, winning 6 extra points (12 total). After each player move their tiles are replenished from a bag that has 108 tiles, game ends when a player runs out of tiles and the bag is empty, that player is given 6 extra points [1]. Backtracking is a programming brute force technique that uses recursion to search for a set of solutions one piece a time, going back in the recursion call stack when a solution fails [2]. The project problem is to create two qwirkle solver algorithms that have backtracking as their foundation, one simple that only takes into account turn points to play and another smarter that decides considering future moves. The solution is implemented in the programming language Python (version 3).

II. Methods

A. Board Data Structure

The core data structure used is the board, is implemented as an expandable two-dimensional array, representing empty positions with zeros and increasing in size when needed when a tile is played. It needs to have always a padding of one empty line in the outside to allow the expected tile moves.

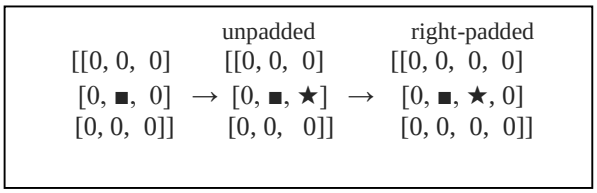

Fig. 1. Board padding example.

B. Backtracking For Valid Combos

The idea for both bots (the solvers) is the same: find all the valid tile combos for a given board and hand of tiles and then apply a filter to pick the best combo. The way of iterating for all time combos is essentially the problem of producing all the permutations of a string, being the string the tile hand with the addition doing an extra iteration for all possible valid positions for each tile. So a combo solution might look as a partial permutation of the hand tile with a position related to each tile. The bare bones idea of the backtracking functionality looks like the following pseudocode:

```
for tile in hand:
    for position in valid positions(tile):
        if board move (tile, position) is valid:
            combo.add(tile, position)
            look a follow-up tile move for a bigger combo
```

Fig. 2. Backtracking pseudocode structure.

One move combos are not discarded against four, or even full hand, combos, because one tile placement might link in multiple directions and scores big points. In the backtracking all tiles in hand are tested as starting points in the combo, this is achieved by holding a running hand index and swapping and un swapping the elements in hand. Each time a bigger combo is searched the board state has to be saved before and restored after the call to delete the possible changes made to it in the nested search loop, this is the basic change-search-restore structure in backtracking algorithms.

Once the backtracking produces all the valid combos all that is left is to pick one. The first bot simply picks the combo that produces the most points, keeping a best combo reference, comparing its points with each one in list and updating it when an option makes more points, from now on we will call this solver points bot.

C. The Smarter Bot

The smarter bot has a refined picking method, it takes two extra aspects: Qwirkle chances caused and possible lines killed. A Qwirkle chance is caused when the combo includes the formation of a 5 tile line, leaving the chance to the next player to make a Qwirke move a score minimum 12 points. The notion of killing a line is when a tile is placed and there is an another tile 1 to 5 empty space away that is incompatible with the placed piece, in a way that a valid line can't be formed with the two as members.

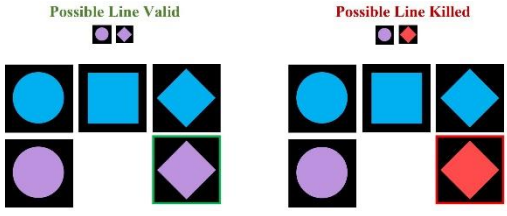

Fig. 3. Line killed example.

These two aspects are calculated for each tile in the combo, the descendent order of importance to pick the smartest combo is: chances, points and lines killed. That is, if two combos have the same points but one leaves the board open to a Qwirkle and the other don't, the last one is the smartest. The lines killed is more of an auxiliary parameter,

since in practice a 7 points combo that kills 2 lines and a 2 points combo that kill not even one, the first is the best candidate to be the smartest one. Starting with the assumption that the first combo is the smartest, the conditions to pick the second as smartest shown next:

```
if chances2 < chances1
elif points2 > points1 and chances2 == chances1
elif points2 == points1 and kills2 <= kills1 and chances2 == chances1
```

Fig. 3. Smarter bot combo picking conditions.

The most important one is the first one, always choose a combo with the minimal chances caused, to score 5 points but leave the next player to do a Qwirkle is a strategy that is not worth it in the long run, especially against the points bot that will make Qwirkle chances for the smarter bot carelessly. A condition solely based on lines killed proved to not be a good reference to better combos, so it's taken as an auxiliary parameter when the other two are equal.

III. ANALYSIS OF RESULTS

Since the smarter bot apply two extra layers of filters is expected to take longer to pick a combo, let's see how it affects real runtime. The following runtimes were done with the same set of tiles for each bot and the same board on the same machine.

TABLE 1. Points vs Smarter Runtimes.

	Points Bot	Smarter Bot
Runtime Seconds	0.0186	0.0467
	0.0245	0.0360
	0.0337	0.0484
	0.0322	0.0333
	0.0280	0.0685
	0.0285	0.0417
	0.0313	0.0577
	0.0319	0.0479
	0.0214	0.0524
	0.0382	0.0412
Average	**0.0288**	**0.0474**

As expected, the smarter bot takes nearly twice the runtime of the simpler bot in average. The function to get the Qwirkle chances is rather linear, is the lines killed function that may be taxing because it has a various while loops nested inside a for loop and has to make quiet the amount of verifications, in theory its impact may not be justified due to his low priority in the combo picking. To better evaluate this aspect here are the results of 10 game sessions played by the bots against each other.

TABLE 2. Points vs Smarter Match.

	Points Bot	Smarter Bot
Match Points	199	205
	176	216
	206	224
	200	197
	198	216
	225	207
	188	207
	221	200
	194	204
	223	202
Advantage Average	2.1	4.8
Wins	**4/10**	**6/10**

The logical hypothesis would be that the smarter bot would surpass the another by a significant amount, a 7 or 6 out of 10 was expected and in practice it was about right. To understand the causes of the smarter bot losing let's analyze the possible aspects.

First, we have the fact that when the combo with most points matches the best option overall, the points bot is actually choosing the best combo in that case. The smarter bot would have to do all the extra analysis to decide the same combo, but this is just a probabilistic situation.

Second, the difference in points that decide who won tend to not be big, like the 194-204 or 200-197, the cases in which the advantage exceeds 40 points are rare, this is expected since both bots have similar foundations. It's important to denote that the bot that finishes the game wins extra 6 points, so looking at the numbers, probably that last points decided the real winner in a couple of these cases.

Third, of course the skill of the player affects the score outcome but the random factor of the bag has big impact on the combos available, even more when taking into account that the bots don't swap tiles with the exception of the extreme rare case when no single move can be done with any hand tile. An important situation to note is that when matching each other the points both had a match ratio of 7-3, very similar to the one against the smarter bot.

For this case it turned out that the added runtime cost of the smarter bot prove itself to make a little difference in winning matches, the most important factor would be the avoid the Qwirkle chances, since they make big boosts in score.

IV. CONCLUSIONS

- The backtracking technique was proved to be an effective way to iterate and accumulate all the valid combos for this board game.
- The Python programming language is very recommended for this problem due to his powerful list functions and syntax, facilitating concise and readable list manipulation code.
- Even though Qwirkle may seem like a really simple game, but big play scenarios revealed that more exceptions were necessary to code, like the restrictions when a combo is 2 or more tiles.

- The fact that the game board is not a static board, like it would be with a game like chess, increased overall complexity of the code because the tile positions were prompted to change, for those cases the id memory addresses of the tiles were used to identify the updated positions.
- To further upgrade the smarter bot the functionality to manage its hand thinking in performing Qwirkles is recommended. Since the current version may use a pair of tiles in a turn that could be used to make a full line the next turn if preserved its winning ratio could increase.

REFERENCES

[1] Mindware, *Qwirkle Rules*, 2011. Accessed on: Oct. 3, 2020. [Online]. Available:https://www.mindware.orientaltrading.com/pdf/instructions/52132.pdf.

[2] D. Matuszek, Backtracking, 2002. Accessed on: Oct. 3, 2020. [Online]. Available:https://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html.