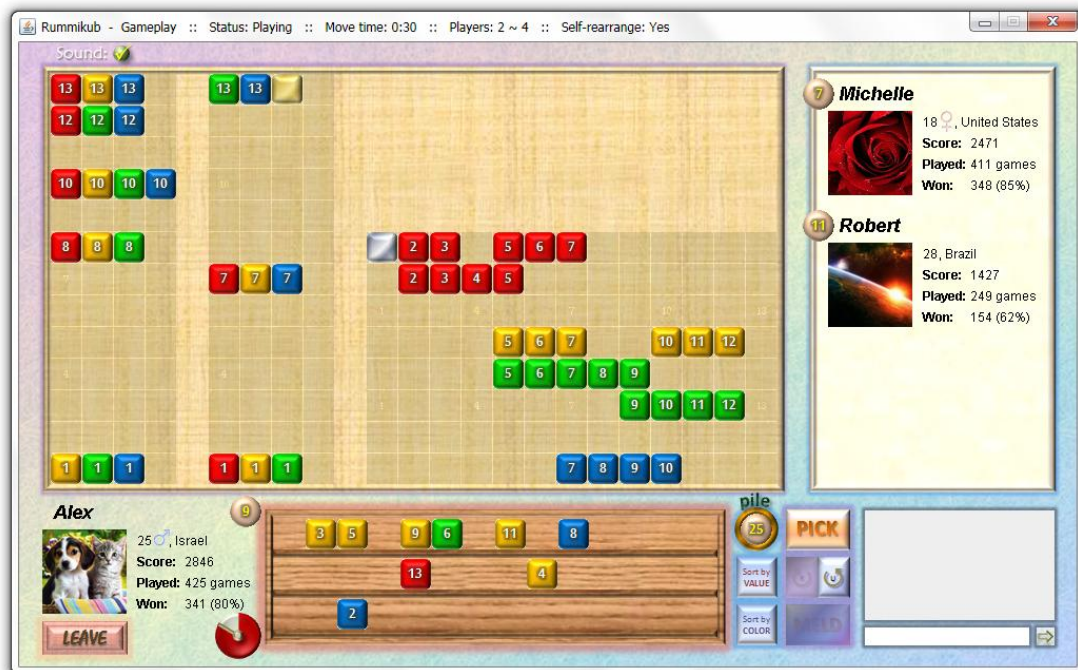


# RUMMIKUB



*Created by:* Alex Kaganovich  
Odeya Bronznick

## Contents:










About Rummikub .....	<a href="#">2</a>
Why Rummikub is a hard problem .....	<a href="#">3</a>
How legal moves are found .....	<a href="#">4</a>
Finding a handful of moves .....	<a href="#">7</a>
Other search methods .....	<a href="#">8</a>
Selecting a move .....	<a href="#">9</a>
About the application .....	<a href="#">10</a>

## **About Rummikub:**

Rummikub (*not* the same as Rummi) is an Israeli board game for 2 to 4 players.

The game consists of card-like tiles: each tile has a number from 1 to 13 and a color in {Red, Yellow, Green, Blue}. There are 2 similar tiles of each kind (equivalent to 2 decks). In addition, there are 2 jokers.

Each player has a rack of tiles visible only to that player, which in the beginning of the game gets 14 tiles. The rest of the tiles are placed in a pile.

The game is based on the concept of **sets** (or **melds**), a horizontal sequence of at least 3 tiles that is either a strictly increasing series of consecutive numbers of the same color (called **run**): [    ], or a sequence of equal numbers without color repetitions (called **group**): [    ]. A set having a joker is legal if there is a non-joker tile that can replace it to make the set legal: [    ].

## **The rules of the game:**

Each player in their turn can extract whatever tiles they like from their rack onto the board and move whatever tiles on the board they like, as long as in the end of the move all the sets on the board are legal. They then declare "meld" and the turn is passed to the next player. If the player is unable to make a legal move, or if they can but choose not to, they pick a tile from the pile and the turn is passed.

Each player is forced to make an initial meld. An initial meld is the first move of the player, and is different from the ordinary moves. Until making their initial meld, and while making it, a player is forbidden to move any tiles extracted to the board by other players and cannot concatenate their own tiles to sets present on the board, and must extract tiles in a total sum of at least 30.

The winner of the game is whoever extracts all their tiles first. The winner's score is the sum of all tiles on all the other players' racks. A joker that is left on the rack is worth 30 points.

A more elaborate explanation about the game can be found [here](#).

A video demonstrating a possible game flow (against human players) can be seen [here](#).

## **Application usage (more on the last page):**

- To start a game, enter as a guest (registration is not possible at this point) and create a room with the desired parameters. You can select the amount and types of robots and whether to play against them or let them play against themselves; the game adds you to the room after the robots, so to do the last, just make the max room capacity equal to the amount of robots.
- Players that have not yet made their initial meld are colored in red.
- The sorting buttons may help you see better what's happening on your rack. The bottom row of the rack isn't sorted, to allow you to store some tiles aside.
- Drag and drop tiles to move. To move many tiles simultaneously, press on one with the right mouse button and drag to select adjacent tiles. Then move them altogether.

## **Why Rummikub is a hard problem:**

The game has many parameters to consider and is very complex for a computer to process.

- Amount of moves per turn: a player that has  $n$  tiles on their rack has at most  $2^n$  move options (where one can think of the empty move as picking from the pile).
- Making a legal move: extracting something from the rack isn't enough; the player should rearrange tiles on the board in order to create legal sets. The time it takes to find a proper rearrangement grows exponentially as the number of tiles on the board increases (if the move is impossible, in some cases there may be no other way to tell but to go over all possible rearrangements).
- Jokers: Jokers can represent any tile and thus greatly increase the number of possibilities to check.
- Hidden information: Given that an opponent has  $n$  tiles and  $m$  is the sum of the tiles in the pile and the amount of tiles of all other players, there are  $\binom{m}{n}$  options for the tiles the opponent has.  
To evaluate the expected reaction of the player, the aforementioned should be calculated for each of those  $\binom{m}{n}$  options.
- Probability: To properly evaluate if picking a tile is worthy, it is needed to traverse all possible tiles the pile may have and calculate the expected result.
- Amount of players: The game isn't limited to only one opponent.

Taking all the above parameters into account is beyond the scope of this project.





Therefore we've mainly focused on finding as many legal moves as possible and choosing the (hopefully) best one, based on rough heuristics.

## How legal moves are found:

Recognizing a legal move is a search problem. However, many of the search methods we've seen won't work here very well because finding a legal move isn't an optimization problem; a move that is "almost legal", with only a few illegal sets, is still illegal and can't be made.

We will use the simple A\* algorithm with an admissible heuristic: it's simple and gets the job done quite well.

First, we define a **tile-set**. Definitions:

- **Set** – a legal sequence of tiles as was previously explained.
- **Partial set** – a set that isn't legal but can be completed into a legal set. For example, [   ] is a partial set whereas [   ] is not. Note that a partial set can be at most 2 tiles long. Further on, as we'll work a lot with these, they'll be referred to as either "sets" or "partial sets", while the original legal sets will be referred to as "legal sets".
- **Tile set** – a collection containing any of the previous two elements.

Second, we define operations on a tile-set. Given a tile-set, one can:

- Extract a tile from the rack.
- Move a tile from one set to another.
- Split a partial set of two tiles into two single-tiled sets.

Each of these operations has a cost of 1.

Next, we define a function  $h(n)$  for evaluating how much a tile-set is far from being legal:

$$h(n) = \frac{\#(partial\ sets)}{2}$$





This heuristic is admissible. How so?

Each of the partial sets should be moved somewhere or something should be moved to it, therefore the minimum amount of operations depends on the number of partial sets, and we divide by 2 because in the best case all the partial sets may be somehow combined in pairs and form a legal tile-set.

The  $g(n)$  function for a tile-set is the number of operations made so far.

Which leads us to defining the  $f(n)$  function for a tile-set, which is:  $f(n) = g(h) + h(n)$ .

### Storing visited states:

Because it's impossible to actually store all board configurations we have already seen (there are  $4!$  different ways to represent the set [     ] alone, that are all equivalent in meaning), we instead store a collection of moved tiles. Once a tile is moved, it is inserted to the collection and cannot be moved again. While not utterly avoiding repetitive states, this does prevent much of them. The maximal number of operations in rearranging the board is defined as MAX\_DEPTH.

The collection of the moved tiles has an additional role in improving performance, which will be explained.

First, we'll sketch the pseudo code for finding *any* move at a turn.

We'll traverse the rack, try to extract the next tile in each iteration (we'll call this loop "outer loop") and try to resolve the resulted tile-set (the resolving loop will be the "inner loop").

Resolving is moving tiles around and/or extracting more tiles in order to get a legal tile-set.

If we succeed we'll return the resolved set, otherwise put the extracted tile back to the rack, extract the next one and try again. If we reach the end of the rack, we'll conclude that no move can be made and pick.

The real board configuration does not change in any time and each iteration of the outer loop relates to the original configuration.

### outer loop (traversing the rack):

```
for each non-joker tile t on the rack {
    extractedTileSet := new tileSet(extract t to the board);
    resolved_set := resolve(extractedTileSet);
    if (resolved_set is successful)
        return resolved_set;
}
return no_move;
```

The inner loop, which is the heart of the search, tries to resolve the tile-set after extracting a tile to the board.

inner loop (resolving a tile-set after extracting a tile):

```
resolve(tileSet) {  
  
    PriorityQueue queue;  
    queue.add(tileSet);  
  
    while (queue is not empty) {  
  
        best := queue.removeBest();  
  
        if (best.legal())  
            return best;  
        if (best.g  $\geq$  MAX_DEPTH or out_of_time)  
            return resolve_unsuccessful;  
  
        illegalSet := best.getShortestIllegalSet();  
  
        //try to concatenate each tile from the illegalSet to every other set  
        for each tile t in illegalSet {  
            for each set s on the board { //except the illegalSet  
                if (concatenate(t, illegalSet).canBeLegal())  
                    queue.add(new TileSet(move(s, illegalSet, t, toLeft)));  
                if (concatenate(illegalSet, t).canBeLegal())  
                    queue.add(new TileSet(move(s, illegalSet, t, toRight)));  
            }  
        }  
    }  
  
    //check which tiles can be added to the illegalSet, on each side  
    canAddOnLeft := whichTilesCanAdd(illegalSet, left);  
    for each tile t in canAddOnLeft {  
        queue.add(new TileSet(concatenate(t, illegalSet)));  
    }  
    canAddOnRight := whichTilesCanAdd(illegalSet, right);  
    for each tile t in canAddOnRight {  
        queue.add(new TileSet(concatenate(illegalSet, t)));  
    }  
  
    //if the illegalSet is 2 tiles long, try splitting it.  
    If (illegalSet.length() == 2) {  
        queue.add(new TileSet(split(illegalSet)));  
    }  
}  
}
```

## **Finding a handful of moves:**

As has been previously noted, the collection of the moved tiles has an additional role: it is also used to not extract tiles from the rack that have already been examined in previous extraction attempts. When extracting a tile from the rack in the resolve function, we put in the moved tiles all the tiles are that are located before this tile on the rack. This way, when attempting to resolve this tile-set, we won't extract tiles that we might have already tried to extract in previous iterations of the outer loop, where the currently extracted tile could have been extracted in that resolve attempt.

Long story short, this is a reasonable alternative to actually storing the board and rack configurations we've previously visited.

Now, having found *some* move, we'll try to find more, preferably all.

For this we will edit the outer loop – after finding a move, we should put it in a collection of possible moves. We will define a collection of currently explored moves (originally containing the initial board configuration), and in a BFS manner we will traverse the collection and in each tile-set there we will try to extract more tiles to get more possible moves.

outer loop (traversing the rack):

```
possibleMoves := empty Collection();
currExplore := empty Collection();
currExplore.add(new TileSet(initialBoardConfiguration));
nextToExplore := empty Collection();

while (currExplore is not empty and time limit has not elapsed) {
  for each tile-set ts in the currExplore {
    for each non-joker tile t on the rack that can be extracted to ts {
      extractedTileSet := new tileSet(extract t to ts);
      resolved_set := resolve(extractedTileSet);
      if (resolved_set is successful) {
        nextToExplore.add(resolvedSet);
        possibleMoves.add(resolvedSet);
      }
    }
  }
  currExplore = nextToExplore;
  nextToExplore = empty Collection();
}
return possibleMoves;
```

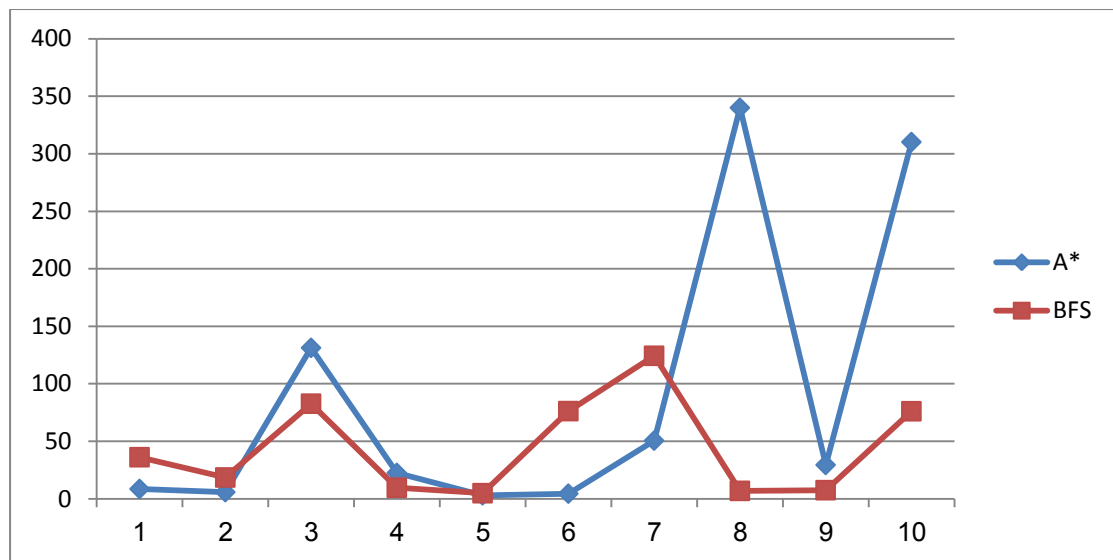
A time limit is set for the search. Overall the robot has 20 seconds to find moves and select one, and the limit in the function above depends on both the overall time limit and the size of *currExplore*. Every tile-set in *currExplore* gets an equal amount of time to be resolved to an additional tile-set.

## Other search methods:

Again, many search methods that suit for problems where nearly optimal solutions are good enough, don't work here. We *could* have tried using some of them, but apart from inflating code that is already of massive quantity, it would probably give notably fewer perfect results (i.e., tile-sets with 0 illegal sets) than A\*, thus be wasteful and unfitting.

The closest opponent for A\* we could think of is the simple BFS without any heuristic. In the experiments to follow, we will choose two robots of different configurations and watch them battle each for 10 games. No experiments involving more than two robots will be conducted. The first experiment involves two **smart** robots (a term that will be explained in the next chapter), one searching for moves with A\* and the other with BFS.

The following results are of two smart robots playing against each other with A\* vs. BFS. Winning percentage was 60% for the benefit of A\*. Number of moves found on average per game round:



As can be seen, results are equivocal. Each method was better in half of the cases.
















BFS didn't need to waste any time calculating heuristic and simply started trying things. It also didn't make a mistake that A\* could, which is to sometimes make a move according to the heuristic, a move that is not on the way to the nearest solution. Most of the time though, following the heuristic was a good idea. But overall, A\* gave a better accumulative result, most notably in games 8 and 10.















## Selecting a move:

No direct way to implement any of the game and tree traversing heuristics we've seen can be directly implemented here. The vast amount of the found moves, the hidden information and the complexity of the moves (especially in later stages) make it nearly impossible to approximate well enough the expected behavior of the opponent. The code has some residue left over from some attempts to take other players' amount of tiles and tiles they extracted throughout the game, but it was commented out. It's just too complex and computationally expensive to evaluate correctly. Thus, we've implemented a **smart robot** that uses a rough heuristic to evaluate the move, based on various factors that a well trained human player may take into account.

First, give a cost for each of these properties (lower costs are better):

- **EXTRACT\_COST = 1**  
Give a cost of 1 to each extracted tile (prefer to extract as less as possible).
- **EXTRACT\_SIMILAR = -100**  
Give a very good value (negative cost = profit) to extracting one of two tiles similar in value and color. For example, if the robot has  , prefer to extract one.
- **AGGREGATE\_SETS\_COST = -500**  
Give an extremely good value to keeping complete sets on the rack instead of extracting them. The value is lower than EXTRACT\_SIMILAR because if the robot has the sets [    ] [    ], then don't extract one of the .
- **EXTRACT\_PARTIAL\_COST = 20**  
Should or should not the robot extract partial sets from the rack.  
Imagine the robot has the partial set [   ]. The robot calculates all the tiles that can complete the set, which are (remember there are two tiles of each kind):    
 . Out of those tiles, it checks how many are present on the board (call the percent of present tiles "ratio"). The value that is given this partial set is: (ratio > 0.5 ? -20 : 20). That is, if the likelihood to extract this partial set in the next round is above 0.5, prefer to keep it.

Regardless of everything mentioned above, any robot:

- Prefers a winning move over any other moves.
- Never make a move that creates a free joker. Example of a tile-set having a free joker: [     ] [    ]. The joker can be replaced with  and available for free use by the player. This kind of moves is really bad.
- The smart robot refrains from creating 4-tile group melds, like [     ]. That's because any of the 8's can be moved from it and used by the next player.
- The smart robot will also keep all its jokers on the rack to the very end. Its playing style produces the best winning percentage but if it loses, it will often lose a lot.
- A robot doesn't pick if it can make a move (that is not a "ייהרג ובל יעבור" move as was mentioned above). It's not always the best thing to do, but it often is.

### Other robots:

Apart from the smart robots, two other robots were created:

- A greedy robot. Attempts to extract in each turn as many tiles as possible (but never creates a free joker). This strategy mimics a naïve player.
- A random robot. Selects a random move among those it found and makes it.

These were created mainly for comparison reasons.

Winning percentage of the smart robot:

- Against the greedy robot: 70%
- Against the random robot: 80%

It could have been interesting to test the smart robot against a human player for 10 games, but sadly no volunteers had the time for that. We did test the robot for a few games against human players (mainly ourselves and several friends) and the winning percentage was:

- Against a human player: 67%

However, many of the games in which the smart robot won, it got a rather low score (the opponent had just one tile on the rack, as can be expected from a greedy player) or the smart robot almost lost (if it didn't win, the opponent would have extracted all their tiles and win). This goes to show that A.I. improvements aren't linear in the amount of effort: to improve the performance just a LITTLE (turn a greedy robot – that uses nearly nothing to evaluate a move – into a smart robot), a LOT of work should be done. Perhaps this is what makes the field of A.I. so deep and intriguing.

## **About the application:**

This game was designed and written from scratch by Alex before the course.  
The A.I. was added later, for the project.

This is a net game that can be played from multiple computers between human players, possibly involving robots.

We were interested in adding an A.I. to the game and had a lot of fun working on it, hopefully you'll find this project interesting and won't get addicted ☺.