I implemented both Monte Carlo Descent algorithm (whose function name is assign\_table\_monte\_carlo(initial\_tables\_guests)) and A star searching algorithm (whose function name is assign\_table\_a\_star(initial\_tables\_guests))to solve task 3. By default, the program will use Monte Carlo Descent when you directly running it. If you want to use A star searching algorithm (I do not recommend so, especially when there are many guests and the relationships are complex… but if you insist…), you should delete the first ‘#’ in the last but four line of my code and comment out the statement above which runs assign\_table\_monte\_carlo(initial\_tables\_guests).

A star searching algorithm is complete and optimal so it’s guaranteed to get the best result. However, I will strongly suggest you use my Monte Carlo function since it not only runs much faster than A star searching algorithm, especially when the total number of guests is quite large, but also it can get the best result most of the time. Even when the total number of guests is quite small, A star searching still has no significant advantages than Monte Carlo Descent.

Generally speaking, there are two reasons why you can always get the best result by using Monte Carlo Descent function: 1st, the problem usually has many local optimizations and they are as good as the best result; 2nd, I runs Monte Carlo Descent several times and choose the best result of them as the final result.

Therefore, I here again strongly suggest that you **Always** use Monte Carlo Descent!

And in the following is a general description of my work.

1. Monte Carlo Descent
2. The problem abstraction is as following:

**State space**: Represented by an M-element vector, where the index of each element corresponds to a certain guest, and each element has a non-negative integer as its value corresponding to the index of the table where the guest are seated. M is the total number of the guests.

**Initial state:** No guests have been seated in any table, represented by an m-element vector where each element is set to 0, which means that the corresponding guest has not been seated in any table yet.

**Goal state:** Every guest has been seated in a table and the number of the tables is as least as possible, represented by an m-element vector where no elements are 0 and the maximum element is as least as possible.

**Cost function:** It equals to the number of the tables.

**Successor function:** Assign a table to a guest who has not been seated yet, given the condition that 1) this new state is never visited before, 2) after assigning, the number of guests in the assigned table will not exceed a given number N, and 3) guests in that table do not know each other before. It is represented by assigning a positive integer P to an element which is currently 0, and making sure that after assigning, 1) the new state is not in our visited state records, 2) the number of P in the vector will not exceed N, and 3) guests with the index that has an element of P do not know each other.

**Edge weights:** It will equal 0 or 1. If after assigning, there will be one more table, then edge weight equals to 1. Otherwise, if the number of tables do not change, the edge weight equals to 0.

**Heuristic function:** It equals to the number of the tables,. Since we directly take the cost function as the heuristic function, it must be admissible.



1. Repeat L times (in my code, L = 10 \* the number of the guests):
2. s = initial state
3. Repeat K times (in my code, K = 10 \* the number of the guests):

* If s is our goal, then return s
* Pick s’ from the successors of s at random
* If h(s’)<=h(s) (we want to find the s’ that makes h(s’) as least as possible!), then s = s’

Else with probability of exp(-(h(s’)-h(s))/T), s = s’

In my code, because h(s’) is either equal to h(s) or 1 greater than h(s), so in this case I can directly replace h(s’)-h(s) by 1.And T = MAX\_TEMPERATURE \* (1 – the times we repeat 2)/K), which makes T decrease as the times we repeat 2) increases.

After repeating 1) and 2) L times, L results will be got. Then the algorithm compares the results with each other based on the number of the tables and chooses the result with the least number of tables as the final result.

1. The problem I met is though Monte Carlo is much faster, it struggles to get the best result sometimes when just running it once. Therefore I repeat it several times and choose the best result as the final result.
2. A star seaching
3. The following problem abstraction is quite similar to Monte Carlo Descent.

**State space**: Represented by an M-element vector, where the index of each element corresponds to a certain guest, and each element has a non-negative integer as its value corresponding to the index of the table where the guest are seated. M is the total number of the guests.

**Initial state:** No guests have been seated in any table, represented by an m-element vector where each element is set to 0, which means that the corresponding guest has not been seated in any table yet.

**Goal state:** Every guest has been seated in a table and the number of the tables is as least as possible, represented by an m-element vector where no elements are 0 and the maximum element is as least as possible.

**Cost function:** It equals to the number of the tables.

**Successor function:** Assign a table to a guest who has not been seated yet, given the condition that 1) this new state is never visited before, 2) after assigning, the number of guests in the assigned table will not exceed a given number N, and 3) guests in that table do not know each other before. It is represented by assigning a positive integer P to an element which is currently 0, and making sure that after assigning, 1) the new state is not in our visited state records, 2) the number of P in the vector will not exceed N, and 3) guests with the index that has an element of P do not know each other.

**Edge weights:** It will equal 0 or 1. If after assigning, there will be one more table, then edge weight equals to 1. Otherwise, if the number of tables do not change, the edge weight equals to 0.

**Heuristic function:** It equals to the number of the tables,. Since we directly take the cost function as the heuristic function, it must be admissible.



1. Add initial state to the fringe. The fringe is implemented by using a priority queue (in Python, it’s called heapq). The priority is generated first based on the number of the tables in the state. If there are two states with the same total number of the tables, then they will be compared based on the order they are added to the fringe.
2. Repeated while fringe is not empty:

* Pop the head h of the fringe
* If h is our goal, return h
* Push the successors of h into the fringe

1. Return False
2. The problem I met is how to use a heuristic way to do the searching. At first I thought I could not directly use the number of the tables as the heuristic function since it’s our cost function and we usually did not directly take the cost function as the heuristic function before. But later on I found for this question the number of the tables just worked well as the heuristic function. So I used it to implement the A star searching.