**INTRODUZIONE**

Ho scelto di rappresentare la scacchiera con una lista di lunghezza N, dove N è il numero delle regine. Gli indici rappresentano le righe, mentre i valori corrispondono alla colonna dove è posizionata la regina di ciascuna riga. Appare evidente il vantaggio che comporta questa scelta, permettendo ad una sola regina di essere posizionata su ciascuna riga. Questo implica che tutte le disposizioni che includono almeno due regine su almeno una stessa riga vengono automaticamente non considerate. Il risparmio, in termini di riduzione della dimensione dello spazio di ricerca, è notevole: da tutte le possibili combinazioni di N\*N numeri presi N volte si passa a N\*\*N possibili stati. Ecco i due spazi a confronto per alcuni valori di N:

|  | comb(N\*N,N) | N\*\*N |
| --- | --- | --- |
| N = 8 | 4’426’165’368 10 | 16’777’216-8 |
| N = 9 | 260’887’834’350-12 | 387’420’489-9 |
| N = 10 | 17’310’309’456’440-14 | 10’000’000’000-11 |
| N = 11 | 1’276’749’965’026’536-16 | 285’311’670’611-12 |
| N = 12 | 103’619’293’824’707’388-18 | 8’916’100’448’256-13 |
| N = 13 | 9’176’358’300’744’339’432-19 | 302’875’106’592’253-15 |

**ALGORITMO BACKTRACKING**

L’algoritmo di ricerca è un *backtracking* con una tecnica di *constraint propagation*. Il constraint propagation è realizzato con una funzione che controlla se la regina nella colonna corrente attacca regine precedentemente posizionate nelle righe superiori, mentre il backtracking si ottiene con una funzione ricorsiva che prova su tutte le colonne della riga corrente e se trova una regina che viene approvata dalla funzione di constraint propagation procede tramite ricorsione alla riga successiva. La ricorsione si interrompe quando viene trovata la prima soluzione.

Lo spazio di ricerca può essere immaginato come un albero completo con *branching factor* = N e profondità = N. Ogni nodo rappresenta un *tentative value* che viene assegnato alla variabile.

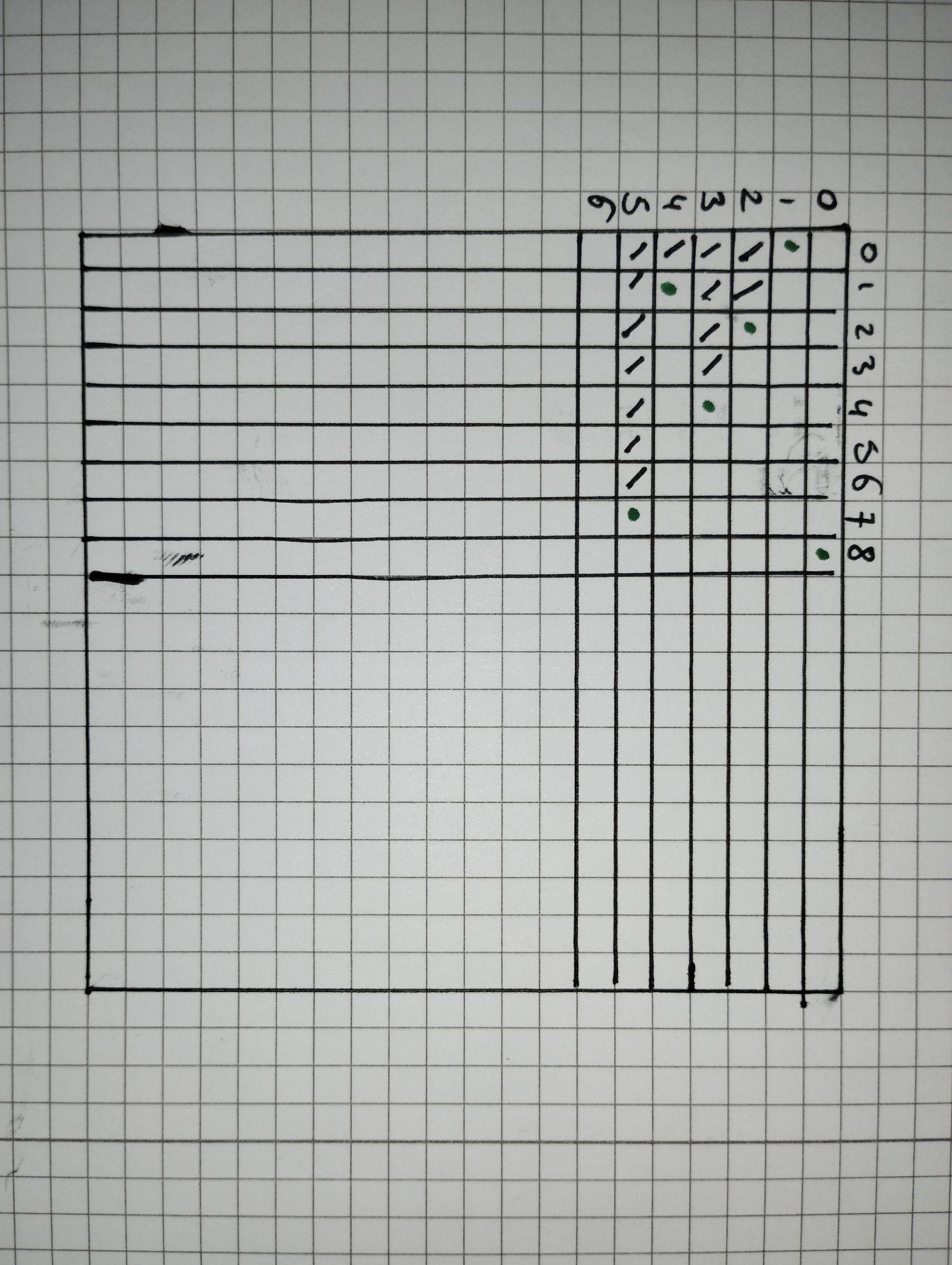
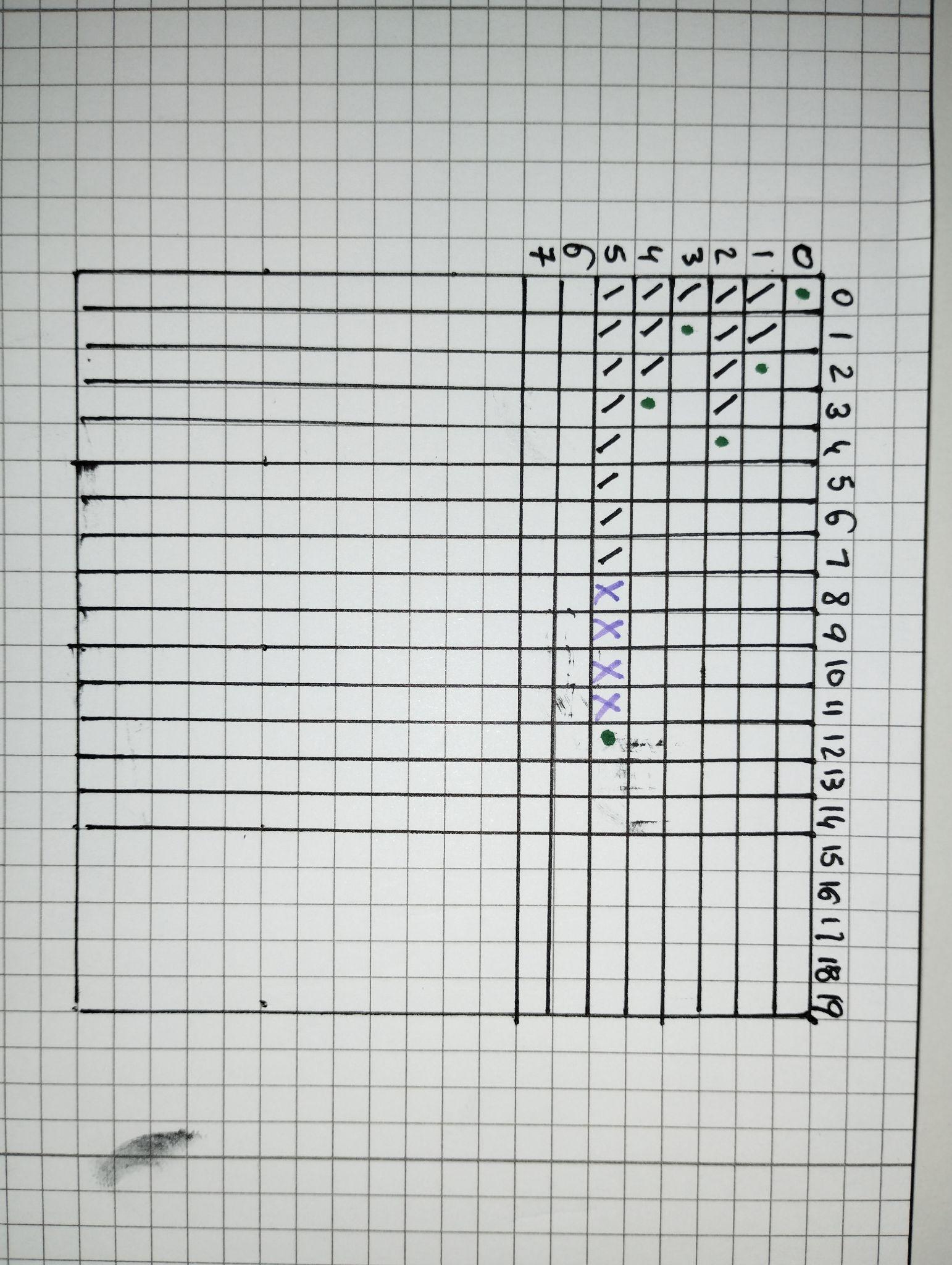
Ciò vuol dire che gli stati possibili sono le foglie dell’albero, ovvero a profondità N quando le N regine sono state posizionate

L’algoritmo procede depth first, quindi non è ottimale; provando a posizionare le regine nelle colonne partendo da quella più a sinistra e procedendo verso destra, si rischia di entrare in recursion successive onerose, o che addirittura non contengono una soluzione, mentre si sarebbe arrivati più in fretta ad una soluzione espandendo il nodo di una colonna più a destra. Ciò provoca una notevole variabilità nelle soluzioni, partendo da colonne diverse si osservano tempi drasticamente ridotti.

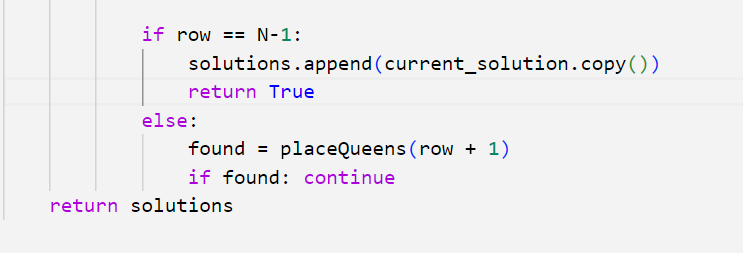
Per esempio, per N=20, se si esegue l’algoritmo partendo dalla prima colonna della prima riga, ci metterà 14s per trovare la prima soluzione ([0, 2, 4, 1, 3, 12, 14, 11, 17, 19, 16, 8, 15, 18, 7, 9, 6, 13, 5, 10]). Mentre se lo si fa partire dalla nona colonna, ci metterà circa 1s per trovare [8, 0, 2, 4, 1, 7, 9, 16, 13, 19, 17, 15, 18, 11, 5, 10, 6, 3, 14, 12].

Nelle figure i pallini rappresentano i nodi che fanno parte della soluzione, le barre nere le posizioni eliminati dalla funzione di constraint propagation , e le croci fucsia i nodi che fanno da radici ad alberi esplorati interamente ma che non hanno ritornato una soluzione. Si può notare come in figura 1.1, partendo dalla prima colonna, arrivati alla sesta riga vengono inutilmente esplorati 4 alberi con 20\*\*15 stati finali ciascuno prima di trovare quello che condurrà alla soluzione, mentre nella figura 1.2, partendo dalla nona colonna, alla stessa riga il primo albero esplorato è quello giusto. Questo spiega la grande variabilità che si osserva nei tempi di esecuzione.

*fig 1.1*  *fig 1.2*



Un approccio alternativo per stimare la complessità temporale è quello di calcolare il tempo di esecuzione per trovare tutte le soluzioni, e dividere per il numero di soluzioni. Per fare esplorare tutti gli stati possibili, basta inizializzare una lista vuota e aggiungere ogni soluzione trovata, e non fare fermare la ricorsione quando se ne trova una.



| N | Tempo (s) | soluzioni | T medio |
| --- | --- | --- | --- |
| 10 | 0.4 | 352 | 5.52E-04 |
| 11 | 1.6 | 2680 | 5.97E-04 |
| 12 | 9.3 | 14200 | 6.55E-04 |
| 13 | 53.2 | 73712 | 7.2E-04 |
| 14 | 346.7 | 365596 | 9.48E-04 |

**SIMULATED ANNEALING**

La funzione obiettivo procede riga per riga, controllando ad ogni iterazione quante regine nelle righe inferiori sono attaccate dalla regina posizionata nella riga corrente. Pertanto, il valore massimo della funzione obiettivo, che si ha per esempio quanto tutte le regine sono nella stessa colonna, è dato da N\*(N-1)/2. Il valore 0 corrisponde ad una soluzione.

L’algoritmo non è completo, infatti potrebbe essere che la funzione si blocchi ad un minimo locale, quando la probabilità di “risalire” è troppo piccola.

Per ovviare a questo problema, lo si può eseguire più volte, e vedere in che percentuale vengono trovate delle soluzioni valide.

Un primo confronto che può essere fatto è quello tra una decrescita lineare ed una esponenziale di T.

Ogni volta che viene eseguito simulated\_annealing, si appende ad una lista il valore della funzione obiettivo dello stato finale, per poi contare quante soluzioni si hanno avuto.

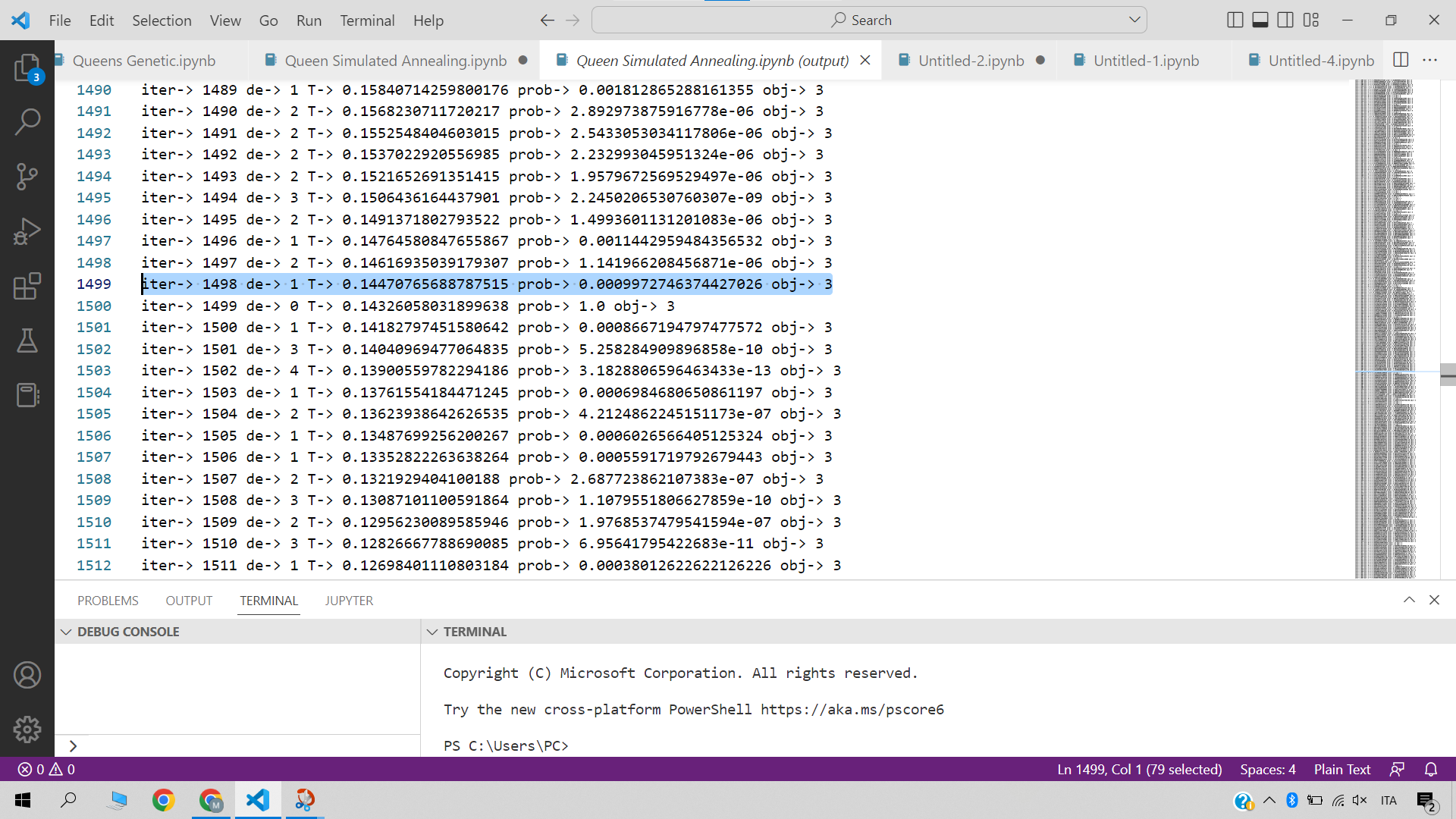
I risultati sotto sono ottenuti con N = 15, il decremento della decrescita lineare = 333, e a, parametro della decrescita esponenziale che viene elevato per l’iterazione corrente, uguale a 0.99.

|  | Iterazione 1 | 2 | 3 | 4 | 5 | Medie |
| --- | --- | --- | --- | --- | --- | --- |
| T lineare soluzioni | 40/50 | 35/50 | 34/50 | 37/50 | 36/50 | 36.5 |
| T lineare tempi | 6.9s | 7.2s | 7.1s | 7.1s | 7.9s | 7.2 |
| T exp soluzioni | 40/50 | 36/50 | 33/50 | 40/50 | 36/50 | 37 |
| T exp  tempi | 6.9s | 7.7s | 7.6s | 7.5s | 7.2s | 7.4 |

I risultati sono molto simili e non si vedono miglioramenti nella performance.

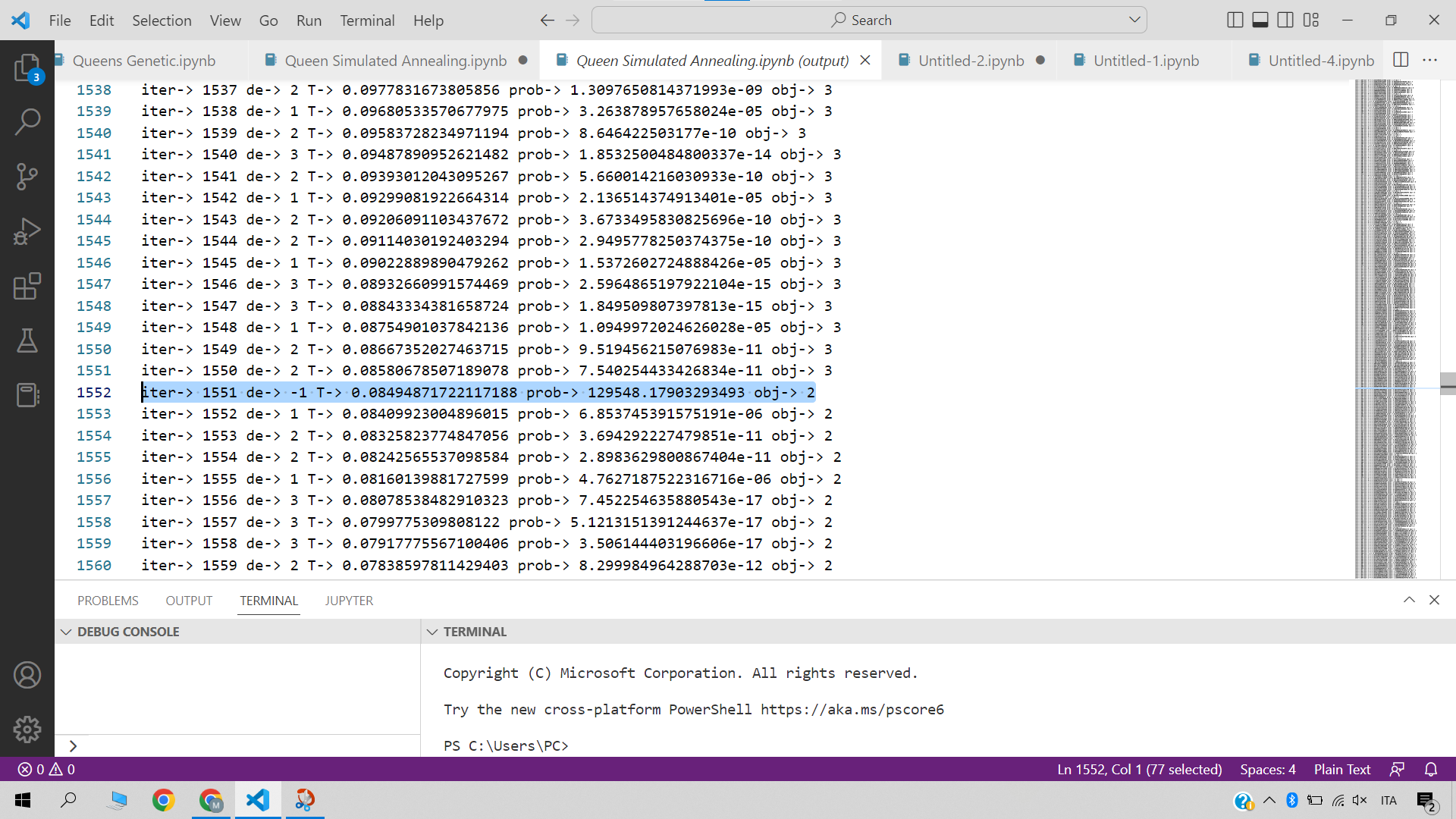
Con i parametri scelti precedentemente, la temperatura raggiunge dopo 1500 iterazioni il valore limite, 0.1447, per cui la probabilità di passare ad uno stato peggiore con de = 1 è di 1/1000.

Si potrebbe pensare che a quell’iterazione si potrebbe fermare il programma, ma un’analisi dell’evoluzione della funzione obiettivo nel tempo rivela che servono molte più iterazioni affinché si arrivi ad una soluzione (bisogna “scendere” fino al minimo globale o locale).

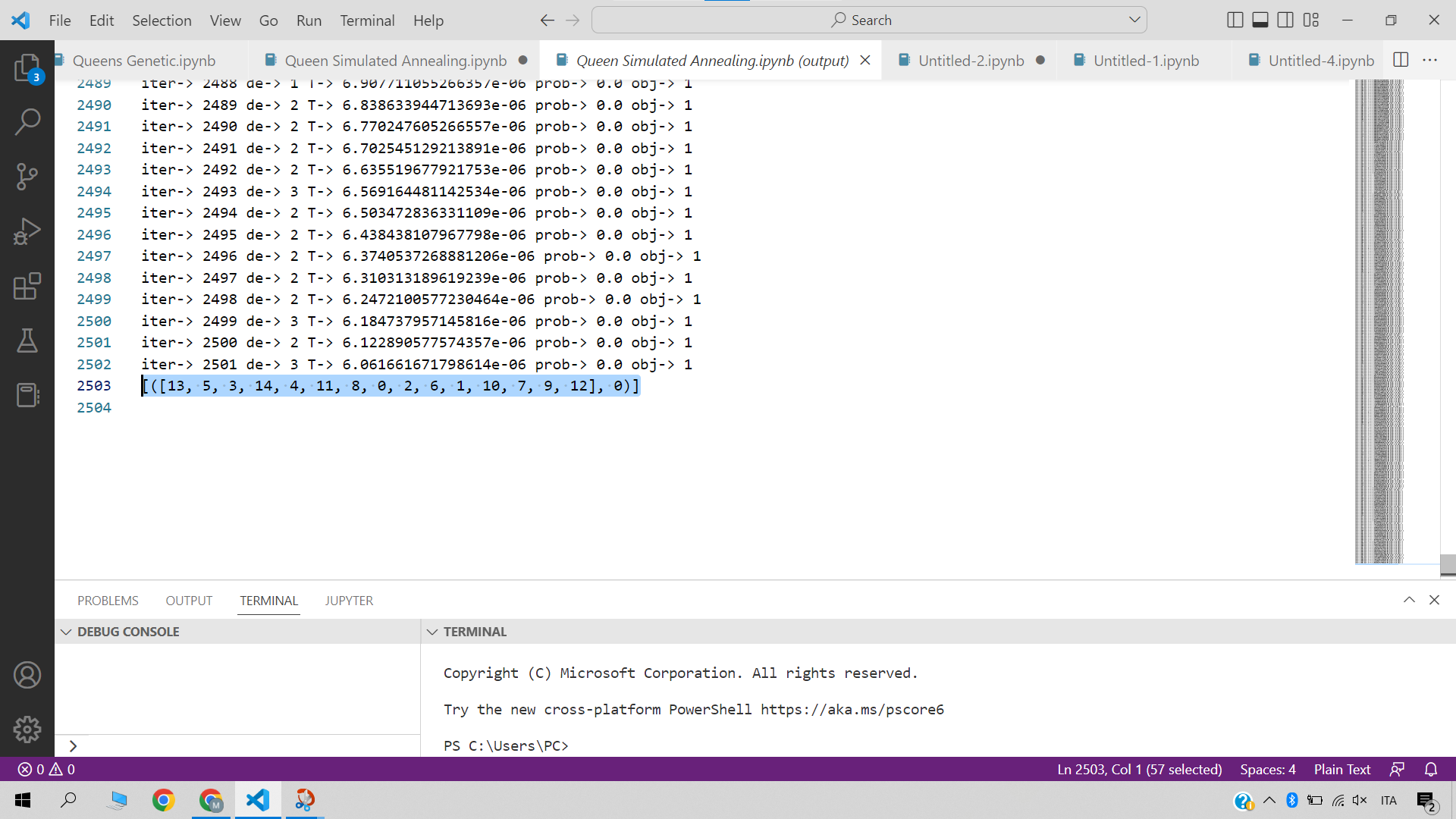
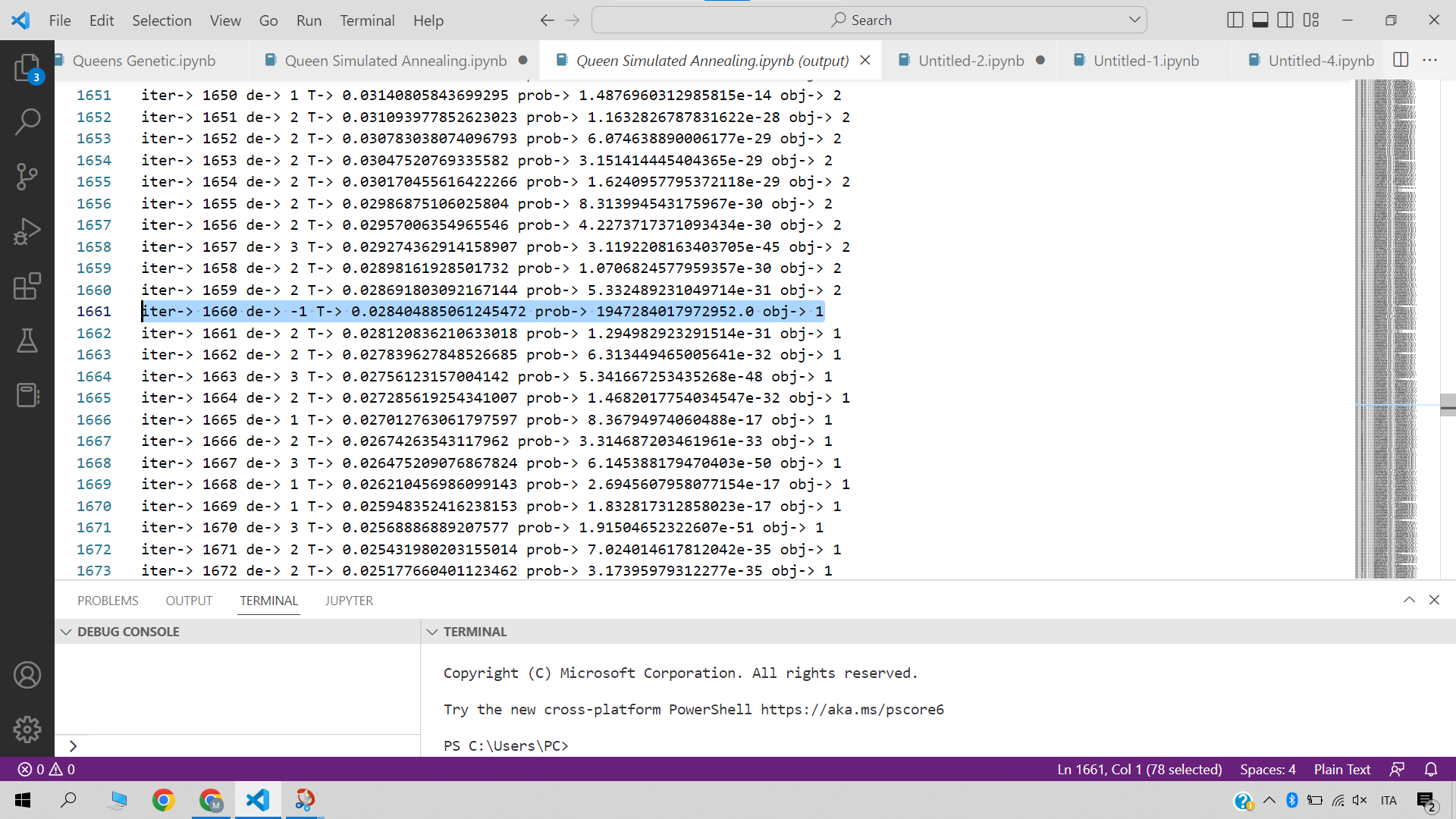


All’iterazione 1498, la T scende sotto la soglia. Da questo momento in poi, non sarà più concesso peggiorare la funzione obiettivo (in ogni caso la probabilità è estremamente bassa).

Eppure, la funzione obiettivo è ancora a tre.



All’iterazione 1551, si trova un vicino migliore (de<0).



La soluzione viene trovata solo all’iterazione 2500.

Qual è un numero ottimale di iterazioni per aumentare la percentuale di successo senza sacrificare troppo in termini di tempo?

Questa volta le iterazioni sono 100, N = 15

|  | 1 | 2 | 3 | 4 | avgs |
| --- | --- | --- | --- | --- | --- |
| 5k-successes | 71 | 77 | 76 | 73 | 75 |
| 5k-times | 13.1s | 13.8s | 14.6s | 14.2s | 14s |
| 10k-successes | 92 | 94 | 89 | 94 | 92 |
| 10k-times | 16.6s | 17.9s | 16.3s | 16.8s | 17s |
| 50k-successes | 100 | 100 | 100 | 100 | 100 |
| 50k-times | 16.8s | 16.9s | 16.9s | 16.8s | 17s |

Due cose si possono notare da questi dati:

-la percentuale 100% delle 50k iterazioni ci dice che se si lascia l’algoritmo girare per abbastanza tempo, con tutta probabilità arriverà ad un minimo globale

-il tempo medio per trovare una soluzione per 5k (14s/75 = 0.1866s) e 10k (17s/100 = 0.17s) sono comparabili, ma questo rapporto inizia ad incrinarsi più N diventa grande (per esempio per N = 20 il rapporto tra i tempi di esecuzione è simile, ma 5k ha successo solo il 45% delle volte mentre 50k continua a mantenere il suo 100%)

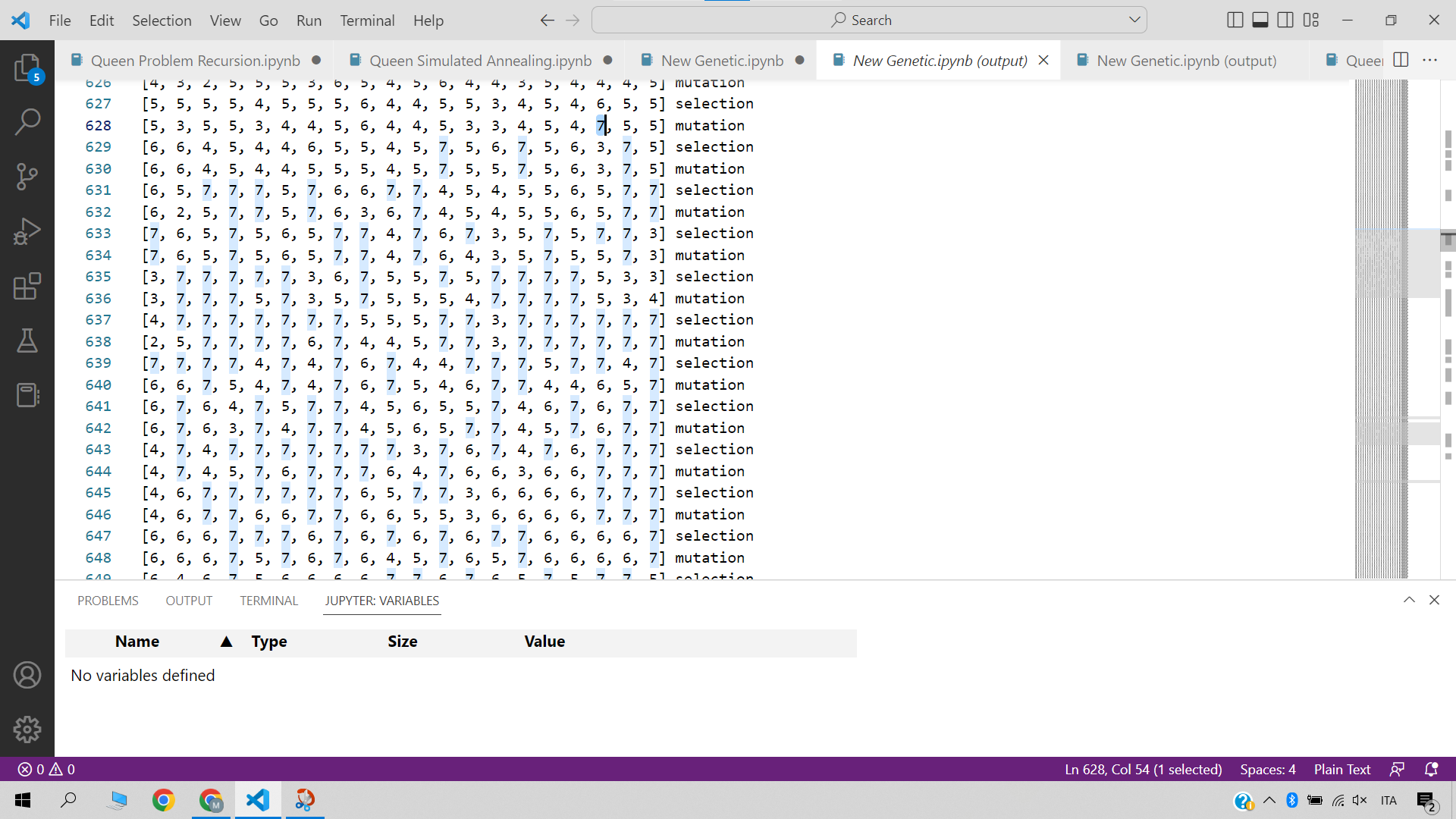
**GENETIC**

Per potere calcolare le probabilità della selezione, ho usato una funzione obiettivo con valori da 0 a N, che conta il numero di regine che non attaccano nessun'altra regina che sta sotto di loro. Quindi una soluzione ha funzione obiettivo = N.

L’algoritmo è formato da tre funzioni in quest’ordine: selezione con roulette wheel, crossover e mutation, tutti con una certa probabilità di esecuzione che è possibile modificare. L’idea di fondo dell’algoritmo genetico è di selezionare gli individui con una funzione obiettivo migliore, pur mantenendo una certa variabilità nella popolazione. Variabilità essenziale, perché in sua assenza le popolazioni tenderanno ad assestarsi su un singolo valore della funzione obiettivo, solitamente non quello di interesse; anche con una percentuale di selezione molto bassa, nel giro di qualche centinaia di iterazioni si afferma il carattere dominante.

Attraverso lo studio dell’evoluzione delle funzioni obiettivo, tentando vari valori dei parametri, si può trovare un giusto mix tra selettività e variabilità che favorisca gli individui con fitness più alta.

La configurazione che ha più successo sembra essere quella con selettività al massimo, perchè deve salvare e moltiplicare più velocemente possibile gli individui con fitness elevata che compaiono da mutazioni o crossover e mutazione e crossover con probabilità che vanno da 0.1 a 0.5 abbastanza alte da impedire che la selezione predomini su individui con fitness bassa ma anche abbastanza bassi da non rendere tutti i tentativi della selezione di salvare gli individui migliori vani.



Questo è un esempio di una propagazione di fitness alte fatta bene.

E’ fatta su una popolazione di 20 individui di cui è riportata una lista con le funzioni obiettivo.

N = 10, la selezione avviene sempre, e la probabilità di mutazione è 0.4.

Alla riga 628, compare un 7 generato da una mutazione. All’iterazione successiva, questo nuovo individuo viene selezionato 3 volte. La mutazione successiva non è abbastanza forte da disperdere le fitness, e l’individuo con fitness 7 continua a propagarsi e mutare aumentando le probabilità che una mutazione generi un 8.

Tuttavia, le prestazioni sono deludenti: anche con N piccoli, l’algoritmo fatica a trovare una soluzione.

Alcuni dati con N = 10 , facendo eseguire l’algoritmo 10 volte e ogni volta con 1000 iterazioni.

|  | 1 | 2 | 3 | 4 | 5 |
| --- | --- | --- | --- | --- | --- |
| Tempo | 20s | 29s | 16s | 18s | 20s |
| Soluzioni | 2/10 | 3/10 | 4/10 | 4/10 | 0/100 |
| Tempo x sol | 10s | 10s | 4s | 4.5 | None |

Anche la dimensione della popolazione aumenta proporzionalmente il tempo di esecuzione ma non porta grossi benefici in termini di soluzioni.

**CONCLUSIONE**

Gli unici veri due contenders sono il backtracking e il simulated annealing. Eppure, il simulated annealing surclassa il backtracking quando N è grande. Per esempio, per N = 27, il backtracking impiega 41s, mentre il simulated annealing impiega in media 2,5s. Inoltre, permette di trovare soluzioni in spazi di ricerca giganteschi con relativa facilità.

Ho eseguito una volta 50 iterazioni del simulated annealing su N=50 e in 5m45s ha reso 48 soluzioni, mentre con il backtracking non mi sono neanche azzardato.

**Considerazioni sulla memoria**

Gli algoritmi non hanno un grande consumo di memoria perchè non ci sono strutture dati complicate. Il numero di liste salvate in memoria per backtracking e simulated annealing è una o al massimo due, perciò hanno consumi di memoria modesti.

Con il modulo tracemalloc, si può vedere il consumo di picco in un dato segmento di codice.

Con N = 15, backtracking ha consumo di memoria di picco pari a 7482 bit. Mentre simulated annealing 3010.

I valori inferiori per simulated annealing possono essere spiegati dal fatto che backtracking deve sempre tenere conto per ogni riga l’ultima colonna che ha esplorato.

Nel caso del genetico, il consumo di memoria è ovviamente più grande e dipende dalla dimensione della popolazione.