Εργασία Peg Solitaire

Σταμάτης Δημήτριος Χατζής (18144)

May 9, 2021

Ο προγραμματισμός έγινε σε περιβάλλον PyCharm, με χρήση της Python 3.8 και σε λειτουργικό σύστημα Windows 10. Μια προσέγγιση σε μια γλώσσα πιο χαμηλού επιπέδου (C/C++) θα είχε ως αποτέλεσμα γρηγορότερο κώδικα, αλλά προτιμήθηκαν οι ευκολίες που προσφέρει η Python με τις βιβλιοθήκες και την εκφραστικότητά της. Επίσης σημαντικό ρόλο έπαιξε και η μεγαλύτερη οικειότητα με την Python (σε σχέση για παράδειγμα με C++).

1 Αναζήτηση πρώτα σε βάθος

Αρχικά υλοποιήσαμε έναν αναδρομικό αλγόριθμο DFS με τη dfs_recursive που αρχικοποιεί το τρέχον μονοπάτι path, υπολογίζει μία φορά ύψος και πλάτος του προβλήματος και ύστερα καλεί την αναδρομική συνάρτηση dfs_recursive_inner.

Η dfs_recursive_inner με τη σειρά της ελέγχει αν έχει επιτευχθεί ο επιθυμητός αριθμός του ενός πασσάλου οπότε τερματίζεται επιτυχώς. Αν υπάρχει επιτρεπτή κίνηση, την εκτελεί και καλεί τον εαυτό της στο νέο στιγμιότυπο, διαφορετικά έχουμε φτάσει σε "κόμβο" του δέντρου αναζήτησης που δεν οδηγεί σε λύση οπότε επιστρέφουμε (backtrack) και στην ουσία καλείται η dfs_recursive_inner στον επόμενο κόμβο του νοητού δέντρου αναζήτησης (σε preorder διάταξη).

Μια προσπάθεια για άρση της αναδρομής και υλοποίηση μη αναδρομικής εκτέλεσης του DFS (συνάρτηση dfs_non_recursive_no_symmetries που αφαιρέθηκε από τον τελικό κώδικα) πέτυχε να κάνει πιο δυσανάγνωστο τον κώδικα χωρίς τα οφέλη στον χρόνο εκτέλεσης. Αν ήταν επιθυμητή η DFS αναζήτηση χωρίς την προϋπόθεση της διατήρησης ίδιας σειράς αναζήτησης με τον αναδρομικό DFS, θα μπορούσαμε να αποφύγουμε την απομνημόνευση της σειράς εμφάνισης για κόμβους του ίδιου επιπέδου.

Επόμενο βήμα ήταν η αποφυγή επανελέγχου ίδιων προβλημάτων ελέγχοντας αν τα συμμετρικά στιγμιότυπα έχουν ήδη συναντηθεί. Συγκεκριμένα: το σύνολο

boards_seen κρατάει όλα τα στιγμιότυπα που έχουν εξεταστεί (και είτε δεν αποτελούν λύση είτε βρισκόμαστε σε ένα απόγονο-στιγμιότυπο στο δέντρο αναζήτησης). Η συνάρτηση symmetries δέχεται ως όρισμα το πρόβλημα και εξετάζει ποιες είναι οι πιθανές συμμετρίες. Οι περισσότερες συμμετρίες που μπορεί να έχει κάποιο ταμπλό είναι 8 (συμμετρίες της διεδρικής ομάδας D_4 τάξεως 8) όταν το πρόβλημα έχει τετραγωνικό σχήμα και οι θέσεις εκτός ταμπλό διατηρούν και τις 8 συμμετρίες.

Οι 4 συμμετρίες που μπορούν να εμφανιστούν μόνο όταν το πρόβλημα έχει διαστάσεις τετραγώνου (λίστα just_square_symmetries) είναι: (α) συμμετρία ως προς την αντιδιαγώνιο, (β) συμμετρία ως προς τη διαγώνιο, (γ) στροφή ωρολογιακά κατά 90 μοίρες και (δ) στροφή αντιωρολογιακά κατά 90 μοίρες.

Οι υπόλοιπες 4 δυνατές συμμετρίες (λίστα orthogonal_symmetries) μπορούν να εμφανιστούν και σε ορθογώνια σχήματα και είναι: (α) ταυτότητα (αυτή είναι και και η μοναδική βέβαιη συμμετρία που υπάρχει σε όλα τα προβλήματα), (β) η κατακόρυφη συμμετρία, (γ) η στροφή κατά 180 μοίρες και (δ) οριζόντια συμμετρία.

Αφού υπολογιστούν οι δυνατές συμμετρίες ελέγχουμε αν έχει ήδη εμφανιστεί κάποιο συμμετρικό του προβλήματος που πρόκειται να εξετάσουμε.

Μια παραλλαγή που δοκιμάστηκε είναι η τυχαία επιλογή ελέγχου επόμενης κίνησης (dfs_recursive_random). Αντί δηλαδή για τη σειρά (0, 1), (1, 0), (0, -1), (-1, 0), κάθε φορά διατάσσουμε τυχαία τις 4 αυτές κατευθύνσεις.

2 Αλγόριθμος πρώτα στο καλύτερο

Επόμενο βήμα ήταν η υλοποίηση του αλγορίθμου πρώτα στο καλύτερο. Μετά από αρκετές δοκιμές και πειράματα με εύκολα υπολογίσιμες συναρτήσεις, καταλήξαμε στις εξής: (α) befs_manhattan: υπολογίζοντας το σύνολο των αποστάσεων Μανχάταν για τα (μη διατεταγμένα) ζεύγη πασσάλων (β) befs_euclidean: υπολογίζοντας το άθροισμα του τετραγώνου των Ευκλείδειων αποστάσεων από το κέντρο (οι θέσεις εκτός ταμπλό δεν συνεισφέρουν στο κέντρο και υπολογίζουμε το 'τετράγωνο' για να μην καλούμε συνεχώς τη συνάρτηση ρίζας) (γ) befs_exponential: με βάση το άθροισμα των "εκθετικών αποστάσεων" των πασσάλων από το κέντρο και (δ) befs_max_exponential: υπολογίζοντας το άθροισμα της μέγιστης ως προς x/y εκθετικής απόστασης από το κέντρο. Πλην της befs_manhattan, όλες οι υπόλοιπες συναρτήσεις της οικογένειας των πρώτα στο καλύτερο, καλούν την befs_generic με όρισμα τη συνάρτηση υπολογισμού κόστους για την ευρετική. Η heapq για την ουρά προτεραιότητας επιλέγει το μικρότερο στοιχείο.

Σκοπός των εκθετικών ευρετικών ήταν να μην προτιμώνται στιγμιότυπα με αρκετούς απομακρυσμένους πασσάλους. Οι τιμές των ευρετικών συναρτήσεων

στα στιγμιότυπα-παιδιά υπολογίζονται με βάση την ευρετική τιμή του στιγμιοτύπου-γονέα για την αποφυγή περιττών υπολογισμών.

Άλλες ευρετικές μέθοδοι που θα μπορούσαν να εξεταστούν σε προβλήματα μεγαλύτερου μεγέθους είναι: η μέση απόσταση των πασσάλων από το ”κέντρο μάζας” του ταμπλό, (β) το πλήθος και η απόσταση των συνεκτικών συνιστωσών μεταξύ τους, (γ) η επιφάνεια της convex hull που περικλείει τους πασσάλους.

3 Πειραματική αξιολόγηση των αλγορίθμων

Ο επεξεργαστής που έτρεξαν τα πειράματα είναι ένας AMD Athlon 64 X2 Dual Core 5200+ με 8GB RAM. Λόγω της παλαιότητας του επεξεργαστή τα προβλήματα προς επίλυση που επιλέχθησαν ήταν σχετικά μικρού μεγέθους.

Το αρχείο experiments.txt περιέχει τα πειράματα που έγιναν και μπορούν να επαναληφθούν (έχοντας υπόψη ότι τα τυχαία προβλήματα που δημιουργούνται θα είναι διαφορετικά) με την εκτέλεση του `peg.py` με μοναδική παράμετρο το `expo`. Για τη δημιουργία τυχαίων επιλύσιμων προβλημάτων χρησιμοποιείται η συνάρτηση create_problem_from_board που δέχεται ως όρισμα ένα ταμπλό με έναν ή κανένα πάσσαλο. Αν δεν υπάρχει πάσσαλος, προσθέτει έναν σε τυχαία επιτρεπτή θέση. Ύστερα, λειτουργώντας με λογική αντίστροφη της επίλυσης προσθέτει πασσάλους μέσω αποδεκτών κινήσεων μέχρι να μην μπορεί να προστεθεί άλλος.

Τα αρχεία που αρχίζουν από “pattern” περιέχουν μοτίβα με βάση τα οποία δημιουργούνται 5 τυχαία ταμπλό. Σε αυτά τα ταμπλό εφαρμόζονται οι συναρτήσεις επίλυσης. Οι λύσεις *δεν* εμφανίζονται στην οθόνη (κάτι τέτοιο είναι δυνατό με τον αποσχολιασμό της σχετικής εντολής), ελέγχονται όμως από τη συνάρτηση validate_solution (εμφανίζεται και σχετικό μήνυμα). Στη συνέχεια ελέγχουμε για λύσεις σε συγκεκριμμένα προβλήματα (αρχεία που το όνομά τους αρχίζει από problem).

Κρίνοντας από τους μέσους όρους των σχετικών (ως προς τον καλύτερο χρόνο για το τρέχον πρόβλημα) χρόνων στο τέλος του αρχείου experiments.txt, βλέπουμε μια υπεροχή του befs_max_exponential. Προτού εκτελέσουμε τα πειράματα είχαμε την πεποίθηση ότι ο befs_exponential θα υπερτερούσε αλλά τελικά φαίνεται να έχει παρόμοιους χρόνους με τον befs_euclidean οπότε η το εκθετικό στοιχείο δεν φαίνεται να βοήθησε σε σχέση με το τετραγωνικό. Το τυχαίο στοιχείο στον dfs_recursive_random για την επιλογή τυχαίας προτεραιότητας στις δυνατές κινήσεις δεν φαίνεται να βοήθησε καθόλου καθώς ακόμα και σε προβλήματα που ο dfs_recursive άργησε πολύ, ο dfs_recursive_random ήταν ακόμα πιο αργός. Ο befs_manhattan είναι πιο αργός από όλους τους αλγορίθμους πρώτα στο καλύτερο.

Η αξιολόγηση μη επιλύσιμων προβλημάτων δείχνει την προφανή υπεροχή του DFS (dfs_recursive) σε προβλήματα χωρίς λύση. Όταν δεν υπάρχει λύση, όλοι οι αλγόριθμοι θα πρέπει να εξετάσουν brute-force όλες τις περιπτώσεις (πλην συμμετρικών). Όμως ενώ στις υλοποιήσεις πρώτα στο καλύτερο, δημιουργούμε νέα αντίγραφα του προβλήματος (συνάρτηση deepcopy για λίστες λιστών), στην υλοποίηση μας του DFS υπάρχει συνεχώς ένα μόνο τρέχον στιγμιότυπο. Έτσι οι απαιτήσεις σε χώρο είναι ελάχιστες αλλά εξίσου σημαντική είναι και η χρονική απόδοση καθώς οι συναρτήσεις flip_pph και flip_hhp είναι πολύ πιο γρήγορες από μία αντιγραφή όλου του ταμπλό που αντιπροσωπεύει ένα πρόβλημα. Με την ευκαιρία να αναφέρουμε ότι αν και στην αρχή χρησιμοποιήσαμε τον τύπο δεδομένων array της numpy, προτιμήσαμε την χρήση μιας λίστας λιστών από ακεραίους για να περιγράψουμε το πρόβλημα, θεωρώντας την ως μία πιο καθαρή λύση σε Python. Πολύ πιθανό όμως η υλοποίηση με array να είναι πολύ πιο αποδοτική στην οικογένεια αλγορίθμων best first search επειδή σημαντικός χρόνος ξοδεύεται στην αντιγραφή λιστών για τη δημιουργία των νέων προβλημάτων.

Σε προβλήματα με αρκετές μη έγκυρες θέσεις, θα ήταν χρήσιμο να μη σαρώνεται εξαρχής το ταμπλό, αλλά να δημιουργείται άπαξ μία λίστα με συντεταγμένες έγκυρων θέσεων και με βάση αυτή να προχωράνε οι αλγόριθμοι.

Τι έχει υλοποιηθεί ενώ δεν ζητήθηκε; Για την ανάγκη επαναληψιμότητας των πειραμάτων και σύγκρισης μεταξύ διαφορετικών υπολογιστών, υλοποιήσαμε δύο απλές συναρτήσεις για την κωδικοποίηση και αποκωδικοποίηση των προβλημάτων (συναρτήσεις encode_board_to_number και decode_number_to_board).

Σε προβλήματα που δεν έχουν λύση, συνήθως ο αλγόριθμος DFS (με έλεγχο των συμμετριών) είναι ο ταχύτερος. Κάτι τέτοιο είναι φυσικό καθώς θα πρέπει να εξεταστούν όλες οι πιθανές κινήσεις. Ο έλεγχος συμμετριών μειώνει δραστικά τον χρόνο εκτέλεσης. Αντίθετα όλες οι παραλλαγές του BeFS ενώ μπορεί να οδηγήσουν πιο γρήγορα σε λύση όταν αυτή υπάρχει αλλά σε περίπτωση αδύνατου προβλήματος οι επιπλέον πράξεις υπολογισμού αυξάνουν τον χρόνο εκτέλεσης.

Αν και επιχειρήσαμε, δεν καταφέραμε να υλοποιήσουμε μια αξιόπιστη λύση για timeout παρά τη χρήση του προτεινόμενου concurrent.futures.

Τρόποι κλήσης του προγράμματος:

```
python peg.py depth input.txt solution.txt
```

Εκτελεί τον αλγόριθμο dfs_recursive στο ταμπλό input.txt και το αποτέλεσμα γράφεται στο solution.txt.

```
python peg.py best input.txt solution.txt
```

Εκτελεί τον αλγόριθμο befs_max_exponential στο ταμπλό input.txt και το αποτέλεσμα γράφεται στο solution.txt.

`python peg.py check input.txt solution.txt`

Ελέγχει για το ταμπλό input.txt τη λύση solution.txt.

`python peg.py expo`

Εκτελεί πειράματα (5 τυχαία προβλήματα για κάθε αρχείο που αρχίζει με pattern και 1 για κάθε αρχείο που αρχίζει με problem) και εμφανίζει πίνακα με τα αποτελέσματα. Από μια τέτοια εκτέλεση προέκυψε το αρχείο experiments.txt. Τα αποτελέσματα εμφανίζονται στην οθόνη και δεν καταγράφονται σε αρχείο.

`python peg.py decode number solution.txt`

Επιλέγει τυχαία έναν από τους αλγορίθμους και τον εκτελεί στο ταμπλό που αντιστοιχεί στην κωδικοποίηση number με τη λύση να γράφεται στο solution.txt.