Προγραμματιστικά Εργαλεία και Τεχνολογίες για Επιστήμη Δεδομένων

Παράδοση 26/9/2019, Νίκος Παπασπύρου.

Εξερεύνηση χώρου καταστάσεων

Ο γρίφος του **λύκου, της κατσίκας και του λάχανου** (γνωστός επίσης και με διαφορετικούς συνδυασμούς ζώων και ζαρζαβατικών) διατυπώνεται ως εξής:

> A farmer wants to cross a river and take with him a wolf, a goat, and a cabbage.
>
> There is a boat that can fit himself plus either the wolf, the goat, or the cabbage.
>
> If the wolf and the goat are alone on one shore, the wolf will eat the goat. If the goat and the cabbage are alone on the shore, the goat will eat the cabbage.
>
> How can the farmer bring the wolf, the goat, and the cabbage across the river?

1. Θέλουμε να γράψουμε ένα πρόγραμμα που θα βρίσκει τη λύση του γρίφου, αντιμετωπίζοντας τον σαν ένα πρόβλημα εξερεύνησης ενός χώρου καταστάσεων. Οι καταστάσεις του προβλήματος περιγράφουν πού βρίσκονται οι τέσσερις πρωταγωνιστές του γρίφου. Στην αρχική κάτασταση, όλοι βρίσκονται στην αριστερή όχθη, και στην τελική (επιδιωκόμενη) κατάσταση, όλοι βρίσκονται στη δεξιά.

2. Θα ορίσουμε μία κλάση `state` για να υλοποιήσουμε τις καταστάσεις.

   ```
   class state:
       def __init__(self, left, right):
           self.left = frozenset(left)
           self.right = frozenset(right)

   init = state(['man', 'cabbage', 'goat', 'wolf'], [])
   ```

   Ο κατασκευαστής της δέχεται δύο παραμέτρους: αυτά που βρίσκονται στην αριστερή όχθη και αυτά που βρίσκονται στη δεξιά. Τα πεδία `left` και `right` των αντικειμένων της κλάσης `state` θα είναι immutable σύνολα (`frozenset`) στα οποία ο κατασκευαστής αναθέτει τις παραμέτρους του.

3. Στη συνέχεια θα ορίσουμε μία μέθοδο `accessible`. Αν `s` είναι μία δοθείσα κατάσταση, τότε `s.accessible()` θα είναι μία γεννήτρια που θα παράγει όλες τις καταστάσεις που είναι "προσβάσιμες" από την `s`. Η κατάσταση `t` είναι προσβάσιμη από την `s` αν η πρώτη μπορεί να προκύψει από τη δεύτερη με μία μόνο μετακίνηση του ανθρώπου (και πιθανώς ενός ακόμη ζώου ή αντικειμένου).

   ```
   def accessible(self):
       if 'man' in self.left:
           for x in self.left:
               moving = frozenset(['man', x])
               yield state(self.left - moving, self.right | moving)
       else:
           for x in self.right:
               moving = frozenset(['man', x])
               yield state(self.left | moving, self.right - moving)
   ```

   Προσέξτε τις πράξεις συνόλων που αναλαμβάνουν τη μετακίνηση από τη μία όχθη στην άλλη των πρωταγωνιστών του γρίφου.

4. Προτού γράψουμε ένα απλό πρόγραμμα που θα επιδεικνύει τη λειτουργία της `accessible`, καλό είναι να υλοποιήσουμε μία μέθοδο που θα αναλάβει την εκτύπωση των καταστάσεων σε μια μορφή εύληπτη για τον άνθρωπο. Χωρίς αυτήν, αν π.χ. στο παραπάνω πρόγραμμα προσθέσουμε:

   ```
   print(init)
   ```

   θα δούμε κάτι σαν το παρακάτω:

   ```
   <__main__.state object at 0x110057be0>
   ```

   που μας δείχνει τον τύπο του αντικειμένου και τη διεύθυνσή του στη μνήμη. Προσθέτουμε λοιπόν την ειδική μέθοδο `__str__`, που αναλαμβάνει τη μετατροπή ενός αντικειμένου σε συμβολοσειρά. Η μέθοδος αυτή καλείται αυτόματα από την `print`.

```
def __str__(self):
    return "left: {}, right: {}".format(
        " & ".join(self.left), " & ".join(self.right)
    )
```

Τώρα η εκτύπωση της αρχικής κατάστασης `init` θα έχει ως αποτέλεσμα:

```
left: cabbage & wolf & man & goat, right:
```

5. Ας δούμε τώρα ποιες καταστάσεις `s` είναι προσβάσιμες από την αρχική και ποιες καταστάσεις `t` είναι προσβάσιμες από αυτές τις `s`, αντίστοιχα:

```
for s in init.accessible():
    print(s)
    for t in s.accessible():
        print("  ", t)
```

Το αποτέλεσμα του παραπάνω θα είναι το εξής:

```
left: wolf & goat, right: cabbage & man
   left: cabbage & wolf & man & goat, right:
   left: wolf & man & goat, right: cabbage
left: cabbage & goat, right: wolf & man
   left: cabbage & wolf & man & goat, right:
   left: cabbage & man & goat, right: wolf
left: cabbage & goat & wolf, right: man
   left: cabbage & goat & man & wolf, right:
left: cabbage & wolf, right: goat & man
   left: cabbage & goat & man & wolf, right:
   left: cabbage & man & wolf, right: goat
```

6. Έχουμε μοντελοποιήσει τις καταστάσεις του προβλήματος και τώρα είμαστε έτοιμοι να εξερευνήσουμε τον χώρο των καταστάσεων για να βρούμε τη λύση του γρίφου. Ξεκινώντας από την αρχική κατάσταση, βρήκαμε προηγουμένως ποιες καταστάσεις είναι προσβάσιμες μετά από δύο μετακινήσεις του ανθρώπου. Το αποτέλεσμα αυτό μας επισημαίνει τα εξής:

   - Δεν έχουμε μοντελοποιήσει κανένα χαρακτηρισμό των καταστάσεων. Δύο ενδιαφέρουσες περιπτώσεις καταστάσεων είναι οι εξής:
     - Η τελική κατάσταση, στην οποία όλοι οι πρωταγωνιστές του γρίφου βρίσκονται στη δεξιά όχθη.
     - Μη ασφαλείς καταστάσεις, στις οποίες είτε ο λύκος και η κατσίκα είτε η κατσίκα και το λάχανο βρίσκονται στη μία όχθη του ποταμού, ενώ ο άνθρωπος βρίσκεται στην άλλη.
   - Δεν έχουμε κάποιο συστηματικό τρόπο εξερεύνησης των καταστάσεων. Δε θα πρέπει να σταματάμε μετά από δύο κινήσεις αλλά να εξερευνούμε οσοδήποτε μεγάλο πλήθος κινήσεων, μέχρι να φτάσουμε στην τελική κατάσταση.
   - Κάποιες κινήσεις είναι άσκοπες. Π.χ. στις πρώτες δύο γραμμές της εκτύπωσης, ο άνθρωπος μεταφέρει το λάχανο στη δεξιά όχθη και στη συνέχεια επιστρέφει με αυτό στην αριστερή όχθη. Ο συνδυασμός των δύο αυτών κινήσεων είναι άσκοπος γιατί η κατάσταση που προκύπτει ως αποτέλεσμα είναι η αρχική. Κατά τη διάρκεια της εξερεύνησης, πρέπει να θυμόμαστε ποιες καταστάσεις έχουμε ήδη επισκεφθεί.

7. Πρώτα θα ορίσουμε τη μέθοδο `success` που θα επιστρέφει `True` αν μία κατάσταση είναι τελική.

```
def success(self):
    return not self.left
```

Η μέθοδος ελέγχει αν δεν υπάρχει κανείς στην αριστερή όχθη. Από αυτό συμπεραίνουμε ότι (εφόσον οι κινήσεις που γίνονται είναι νόμιμες) όλοι οι πρωταγωνιστές του γρίφου βρίσκονται στη δεξιά όχθη.

8. Στη συνέχεια θα ορίσουμε τη μέθοδο `safe` που θα επιστρέφει `True` αν μία κατάσταση είναι ασφαλής.

```
def safe(self):
    def tragic(bank):
        BAD = [('cabbage', 'goat'), ('wolf', 'goat')]
        return 'man' not in bank and \
                any(frozenset(pair) <= bank for pair in BAD)
    return not tragic(self.left) and not tragic(self.right)
```

Εδώ ελέγχουμε αν συμβαίνει κάτι τραγικό (συνάρτηση `tragic`) είτε στην αριστερή είτε στη δεξιά όχθη του ποταμού. Το τραγικό θα είναι ο άνθρωπος να μη βρίσκεται σε αυτή την όχθη αλλά να βρίσκεται εκεί ένα από τα "κακά" ζευγάρια του γρίφου, δηλαδή είτε η κατσίκα και το λάχανο είτε ο λύκος και η κατσίκα. Προσέξτε τον τελεστή `<=` που συμβολίζει τη σχέση υποσυνόλου ($\subseteq$), όταν εφαρμόζεται σε σύνολα.

9. Τώρα μπορούμε να συνδυάσουμε τις δύο αυτές μεθόδους με την εξερεύνηση δύο κινήσεων που γράψαμε, για να αποφύγουμε τις μη ασφαλείς καταστάσεις και για να σταματήσουμε αν η εξερεύνηση μάς οδηγήσει στην τελική κατάσταση (που δυστυχώς δε θα συμβεί τόσο εύκολα).

```
for s in init.accessible():
    if not s.safe(): continue
    print(s)
    for t in s.accessible():
        if not t.safe(): continue
        print("  ", t)
        if t.success(): print("We're finished!")
```

Το αποτέλεσμα του παραπάνω θα είναι τώρα το εξής:

```
left: cabbage & wolf, right: goat & man
   left: goat & cabbage & man & wolf, right:
   left: cabbage & man & wolf, right: goat
```

Αυτό σημαίνει ότι η πρώτη κίνηση είναι υποχρεωτική: ο άνθρωπος πρέπει να μεταφέρει την κατσίκα στη δεξιά όχθη. Στη συνέχεια μπορεί είτε να επιστρέψει με την κατσίκα (αργότερα θα αποκλείσουμε αυτή την κίνηση ως άσκοπη) ή να επιστρέψει μόνος του.

10. Προτού προχωρήσουμε στην πιο συστηματική εξερεύνηση του χώρου των καταστάσεων, θα χρειαστεί να ασχοληθούμε με τη σχέση ισότητας των καταστάσεων. Πότε δύο καταστάσεις είναι ίσες; Οι καταστάσεις είναι αντικείμενα στην Python και την απάντηση σε αυτή την ερώτηση πρέπει να τη δούμε γενικότερα.

Αν `x` και `y` είναι δύο αντικείμενα, ο έλεγχος ισότητας `x != y` καλεί αυτόματα την ειδική μέθοδο `__eq__` του αντικειμένου `x`, περνώντας ως παράμετρο το αντικείμενο `y`. (Το ίδιο συμβαίνει και με τους άλλους τελεστές σύγκρισης, που έχουν αντίστοιχες μεθόδους.) Για τα προκαθορισμένα αντικείμενα της Python (π.χ. `int`, `str`, λίστες, σύνολα, κ.λπ.), η μέθοδος αυτή είναι ορισμένη να συμπεριφέρεται όπως κανείς περιμένει (π.χ. δύο σύνολα είναι ίσα όταν έχουν ακριβώς τα ίδια στοιχεία). Για αντικείμενα που ορίζουμε εμείς, όπως αυτά της κλάσης `state` παραπάνω, η default μέθοδος `__eq__` κάνει κάτι πολύ απλό: συγκρίνει το `id` των αντικειμένων (δηλαδή τη διεύθυνσή τους) στη μνήμη. Επομένως, αν έχουμε:

```
init1 = state(['man', 'cabbage', 'goat', 'wolf'], [])
init2 = state(['man', 'cabbage', 'goat', 'wolf'], [])

print("init1 has id", id(init1))
print("init2 has id", id(init2))
print(init1 == init2)
```

το αποτέλεσμα της σύγκρισης θα είναι `False` γιατί πρόκειται για δύο διαφορετικά αντικείμενα (δηλαδή δύο αντικείμενα που κατασκευάστηκαν με δύο διαφορετικές κλήσεις του κατασκευαστή):

```
init1 has id 4426181152
init2 has id 4426181264
False
```

Αυτό φυσικά δεν μας εξυπηρετεί, γιατί στο μυαλό μας οι δύο καταστάσεις `init1` και `init2` είναι ίσες, εφόσον οι πρωταγωνιστές του γρίφου βρίσκονται στις ίδιες θέσεις και στις δύο.

11. Για να το διορθώσουμε αυτό, ορίζουμε την ειδική μέθοδο `__eq__` της κλάσης `state` ως εξής:

```
def __eq__(self, other):
    return self.left == other.left
```

Δύο καταστάσεις είναι ίσες αν και μόνο αν τα σύνολα όσων βρίσκονται στην αριστερή όχθη είναι ίσα. Το αποτέλεσμα της παραπάνω σύγκρισης τώρα θα είναι `True`, παρότι τα `id` των δύο αντικειμένων θα είναι διαφορετικά.

12. Προσέξτε ότι στην Python, εν αντιθέσει με τις γλώσσες που διαθέτουν στατικό σύστημα τύπων, οποιαδήποτε δύο αντικείμενα μπορούν να συγκριθούν μεταξύ τους, π.χ. το αποτέλεσμα της έκφρασης `[3] == "hello"` είναι `False`. Η μέθοδος `__eq__` που ορίσαμε παραπάνω θα οδηγήσει σε εξαίρεση, αν το δεύτερο αντικείμενο (`other`) δεν είναι

κατάσταση (για την ακρίβεια, αν δε διαθέτει πεδίο `left`). Για να επιτρέπουμε τη σύγκριση και με αντικείμενα άλλων κλάσεων, θα ήταν καλύτερα να ορίσουμε την `__eq__` ως εξής:

```
def __eq__(self, other):
    return isinstance(other, state) and self.left == other.left
```

Η συνάρτηση `isinstance` ελέγχει αν η πρώτη παράμετρος είναι ένα αντικείμενο προερχόμενο από την κλάση που δίνεται στη δεύτερη παράμετρο.

13. Για τη συστηματική εξερεύνηση του χώρου των καταστάσεων θα εφαρμόσουμε την τεχνική της αναζήτησης κατά πλάτος (breadth-first search, BFS). Η βασική ιδέα είναι ότι χρησιμοποιούμε μία **ουρά**, στην οποία αρχικά τοποθετούμε την αρχική κατάσταση. Στη συνέχεια, όσο η ουρά δεν είναι άδεια, αφαιρούμε το πρώτο στοιχείο της ουράς, βρίσκουμε τις καταστάσεις που είναι προσβάσιμες από αυτήν και τις προσθέτουμε στο τέλος της ουράς. Με τον τρόπο αυτό εξασφαλίζουμε ότι θα επισκεφθούμε όλες τις καταστάσεις με τις ελάχιστες δυνατές κινήσεις, ξεκινώντας από την αρχική.

    Ορίζουμε μία συνάρτηση `solve` που εφαρμόζει τον αλγόριθμο του BFS όπως παραπάνω. Για την υλοποίηση της ουράς, χρησιμοποιεί την κλάση `deque` που ορίζεται στο module `collections` της βιβλιοθήκης της Python. Η συνάρτηση `solve` θα επιστρέψει την τελική κατάσταση, αν τη βρει. Αν αντίθετα η ουρά αδειάσει χωρίς να φτάσουμε στην τελική κατάσταση, τότε η συνάρτηση θα επιστρέψει χωρίς κάποια τιμή (δηλαδή θα επιστρέψει την τιμή `None`).

    ```
    from collections import deque

    def solve():
        init = state(['man', 'cabbage', 'goat', 'wolf'], [])
        Q = deque([init])
        while Q:
            s = Q.popleft()
            for t in s.accessible():
                if t.success():
                    return t
                if t.safe():
                    Q.append(t)

    print(solve())
    ```

    Το πρόγραμμά μας τώρα είναι σε θέση να βρει την τελική κατάσταση. Εκτυπώνει:

    ```
    left: , right: goat & man & wolf & cabbage
    ```

14. Στη συστηματική αναζήτηση του χώρου των καταστάσεων έχουμε παραβλέψει μία σημαντική λεπτομέρεια. Το παραπάνω πρόγραμμα επισκέπτεται την ίδια κατάσταση πολλές φορές. Για παράδειγμα, με δύο κινήσεις μπορεί να προκύψει και πάλι η αρχική κατάσταση αν ο άνθρωπος φύγει με την κατσίκα στην πρώτη κίνηση και επιστρέψει μαζί της στη δεύτερη. Αυτό έχει ως αποτέλεσμα να εξερευνούμε πολλές φορές τις ίδιες καταστάσεις — η εύρεση της λύσης καθυστερεί πολύ. Ο μόνος λόγος που το πρόγραμμά μας λειτουργεί σωστά είναι ότι για αυτόν το γρίφο ο χώρος καταστάσεων είναι πολύ μικρός (όλες οι δυνατές καταστάσεις είναι 16 και οι ασφαλείς είναι μόνο 10).

    Για να περιορίσουμε την αναζήτηση έτσι ώστε να επισκεπτόμαστε κάθε κατάσταση ακριβώς μία φορά, πρέπει να προσθέτουμε όλες τις καταστάσεις που έχουν προστεθεί στην ουρά (ανεξαρτήτως αν έχουν αργότερα αφαιρεθεί) σε ένα σύνολο `seen` και να μην ξαναπροσθέτουμε στην ουρά κάποια κατάσταση που ανήκει ήδη στο σύνολο `seen`. Η προσθήκη του συνόλου των καταστάσεων που έχουμε επισκεφθεί είναι βασικό συστατικό του αλγορίθμου BFS.

    ```
    def solve():
        init = state(['man', 'cabbage', 'goat', 'wolf'], [])
        Q = deque([init])
        seen = set([init])
        while Q:
            s = Q.popleft()
            for t in s.accessible():
                if t.success():
                    return t
                if t.safe() and t not in seen:
                    Q.append(t)
                    seen.add(t)
    ```

15. Δυστυχώς το πρόγραμμά μας τώρα δεν τρέχει. Αποτυγχάνει με το εξής μήνυμα:

```
Traceback (most recent call last):
  File "./2.py", line 58, in <module>
    print(solve())
  File "./2.py", line 48, in solve
    seen = set([init])
TypeError: unhashable type: 'state'
```

Το μήνυμα αυτό λέει ότι το αντικείμενο `init` που ανήκει στην κλάση `state` δεν μπορεί να τοποθετηθεί στο σύνολο `seen` γιατί ο τύπος (η κλάση) `state` δεν είναι "hashable". Τα σύνολα στην Python υλοποιούνται με πίνακες κατακερματισμού (hash tables), δομές δεδομένων που εξασφαλίζουν σχεδόν σταθερό κόστος αναζήτησης, εισαγωγής και διαγραφής. Για να μπορεί κάποιο στοιχείο να εισαχθεί σε έναν πίνακα κατακερματισμού θα πρέπει να ορίζεται για αυτό η τιμή κάποιας συνάρτησης κατακερματισμού (hash function), που θα επιστρέφει για αυτό το στοιχείο έναν ακέραιο αριθμό. Στην Python, αυτό μπορεί να γίνει ορίζοντας για την κλάση `state` την ειδική μέθοδο `__hash__`, π.χ. ως εξής:

```
def __hash__(self):
    return 42         # this is a very bad idea!
```

Με τον τρόπο αυτό, η τιμή της συνάρτησης κατακερματισμού για κάθε κατάσταση είναι ίδια και ίση με 42. Αυτό δουλεύει μεν, αλλά είναι πολύ κακή ιδέα.[1] Η συνάρτηση κατακερματισμού είναι καλό να επιστρέφει διαφορετικές τιμές για διαφορετικές καταστάσεις. Ο ευκολότερος τρόπος να την ορίσουμε σωστά στην Python είναι να χρησιμοποιήσουμε την προκαθορισμένη συνάρτηση `hash` που ορίζεται για όλους τους βασικούς τύπους της Python και να την καλέσουμε για το πεδίο `left` της κατάστασης (ένα `frozenset`). Δεν είναι τυχαίο ότι επιλέξαμε ακριβώς αυτό το πεδίο που συμμετέχει στον έλεγχο ισότητας καταστάσεων.

```
def __hash__(self):
    return hash(self.left)
```

Τώρα, το πρόγραμμα θα έχει το ίδιο αποτέλεσμα με το προηγούμενο, δηλαδή οδηγείται στην τελική κατάσταση και την εκτυπώνει. Όμως, επειδή αποφεύγει να επισκέπτεται ξανά καταστάσεις που έχει ήδη δει, τοποθετεί στην ουρά συνολικά μόνο 9 καταστάσεις, αντί 113 καταστάσεων του προηγούμενου προγράμματος! Σε κάποιο πρόβλημα με μεγαλύτερο χώρο καταστάσεων, η διαφορά θα ήταν τεράστια και το πρώτο πρόγραμμα πρακτικά δε θα τερμάτιζε ποτέ.

16. Το τελευταίο μας θέμα, τώρα που υλοποιήσαμε σωστά το BFS, είναι ότι το πρόγραμμά μας εκτυπώνει την τελική κατάσταση, δε μας λέει όμως με ποια σειρά κινήσεων οδηγηθήκαμε εκεί, δηλαδή δε μας δίνει τη λύση του προβλήματος.

    Για πάρουμε τη λύση του προβλήματος πρέπει να γνωρίζουμε για κάθε κατάσταση την οποία επισκεπτόμαστε ποια ήταν η προηγούμενή της, δηλαδή ποια κίνηση μας οδήγησε σε αυτήν. Υπάρχουν (τουλάχιστον) δύο τρόποι να αποθηκεύσουμε αυτή την πληροφορία:

    - Να την προσθέσουμε μέσα στις καταστάσεις: κάθε κατάσταση θα περιέχει ως επιπλέον πεδία την προηγούμενη κατάσταση (ή `None`, στην περίπτωση της αρχικής) και πιθανώς την κίνηση που μας οδήγησε εκεί (αν και θα μπορούσαμε εύκολα να την υπολογίζουμε).
    - Να μετατρέψουμε το σύνολο `seen` σε ένα λεξικό, το οποίο για κάθε κατάσταση που έχουμε επισκεφθεί θα μας δίνει την προηγούμενή της κατάσταση.

    Στην παράδοση αποφασίσαμε να υλοποιήσουμε τη δεύτερη προσέγγιση. (Στη σελίδα του μαθήματος μπορείτε να δείτε το πλήρες πρόγραμμα και για την πρώτη προσέγγιση.) Η συνάρτηση `solve` και η κλήση της γίνονται τώρα:

```
def solve():
    init = state(['man', 'cabbage', 'goat', 'wolf'], [])
    Q = deque([init])
    seen = {init: None}
    def solution(s):
        t = seen[s]
        if t is None: return [s]
        return solution(t) + [s]
    while Q:
        s = Q.popleft()
        for t in s.accessible():
```

[1] Αν διαβάσετε πώς λειτουργεί ένας πίνακας κατακερματισμού, θα καταλάβετε ότι με αυτή τη συνάρτηση όλες οι καταστάσεις θα προστίθενται στο ίδιο κελί του πίνακα και άρα αντί για σύνολο θα έχουμε στην πραγματικότητα μία λίστα — το κόστος αναζήτησης, εισαγωγής και διαγραφής θα είναι γραμμικό, αντί σχεδόν σταθερό.

```
            if t.success():
                seen[t] = s
                return solution(t)
            if t.safe() and t not in seen:
                Q.append(t)
                seen[t] = s

for s in solve():
    print(s)
```

Η εντολή `seen = {init: None}` ορίζει την αρχική τιμή του λεξικού. Περιέχει μόνο το κλειδί `init` (δηλαδή την αρχική κατάσταση), η οποία δεν έχει προηγούμενη (η τιμή που της αντιστοιχεί είναι `None`). Όταν επισκεφτόμαστε μία νέα κατάσταση, προσθέτουμε στο λεξικό την προηγούμενή της με την εντολή `seen[t] = s`. Προσέξτε τον έλεγχο `t not in seen`, που επιστρέφει `True` αν η κατασταση `t` δεν υπάρχει ως κλειδί στο λεξικό `seen`.

Η συνάρτηση `solution(s)` αναλαμβάνει να επιστρέψει τη λίστα των καταστάσεων που οδηγούν στην κατάσταση `s` ξεκινώντας από την αρχική. Χρησιμοποιεί τις τιμές που έχουν αποθηκευθεί στο λεξικό `seen` και είναι αναδρομική. Τη γράφουμε έτσι γιατί γνωρίζουμε ότι η λίστα των κινήσεων είναι σχετικά μικρή — αν ήταν μεγαλύτερη θα έπρεπε να τη γράψουμε αποδοτικότερα.

Το πρόγραμμά μας τώρα εκτυπώνει τη σειρά των καταστάσεων από την αρχική μέχρι και την τελική:

```
left: goat & man & cabbage & wolf, right:
left: cabbage & wolf, right: goat & man
left: man & cabbage & wolf, right: goat
left: wolf, right: goat & man & cabbage
left: goat & man & wolf, right: cabbage
left: goat, right: man & cabbage & wolf
left: goat & man, right: cabbage & wolf
left: , right: goat & man & cabbage & wolf
```

NumPy

1. Το NumPy είναι μία εξαιρετική βιβλιοθήκη της Python για επιστημονικούς υπολογισμούς. Μεταξύ άλλων υποστηρίζει:

   - έναν αποδοτικό τύπο για N-διάστατους πίνακες
   - δυνατότητα ενσωμάτωσης συναρτήσεων γραμμένων σε C/C++ ή Fortran
   - χρήσιμες συναρτήσεις και αλγορίθμους (γραμμική άλγεβρα, μετασχηματισμός Fourier, γεννήτρια τυχαίων αριθμών, κ.λπ.)

2. Για να το χρησιμοποιήσουμε, βεβαιωνόμαστε πρώτα ότι υπάρχει εγκατεστημένο στον υπολογιστή μας (το `numpy` δεν είναι πακέτο της βασική βιβλιοθήκης της Python).

   Συνήθως, το κάνουμε `import` και το μετονομάζουμε σε κάτι συντομότερο, π.χ. `np`, για διευκόλυνσή μας.

   ```
   import numpy as np
   ```

3. Το NumPy υποστηρίζει ομογενείς πολυδιάστατους πίνακες (arrays), πολύ αποδοτικότερους από τις λίστες της Python. Παρακάτω φαίνεται εν τάχει ο τρόπος ορισμού και η βασική τους χρήση. Πρώτα ένας μονοδιάστατος πίνακας (διάνυσμα):

   ```
   >>> a = np.array([1, 2, 3])
   >>> print(a)
   [1 2 3]
   >>> print(type(a))
   <class 'numpy.ndarray'>
   >>> print(a.ndim)
   1
   >>> print(a.shape)
   (3,)
   >>> print(a[1])
   2
   ```

   Προσέξτε ότι τα arrays είναι mutable και ότι η αρίθμηση των στοιχείων ξεκινάει από το μηδέν:

```
>>> a[1] = 42
>>> print(a)
[ 1 42  3]
```

Στη συνέχεια ένας διδιάστατος πίνακας:

```
>>> b = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(b)
[[1 2 3]
 [4 5 6]]
>>> print(type(b))
<class 'numpy.ndarray'>
>>> print(b.ndim)
2
>>> print(b.shape)
(2, 3)
>>> print(b[0][1])
2
>>> print(b[0, 1])
2
>>> b[0, 1] = 17
>>> print(b)
[[ 1 17  3]
 [ 4  5  6]]
```

Τα πεδία `ndim` και `shape` των arrays μας δίνουν το πλήθος των διαστάσεων και το μέγεθος των πινάκων, αντίστοιχα. Προσέξτε ότι στον διδιάστατο πίνακα, τα `b[0][1]` και `b[0, 1]` είναι ισοδύναμα — αναφέρονται στο δεύτερο στοιχείο της πρώτης γραμμής. Για την ακρίβεια, το `b[0]` αναφέρεται στην πρώτη γραμμή του πίνακα `b`:

```
>>> print(b[0])
[ 1 17  3]
>>> print(b[0].shape)
(3,)
```

4. Οι πίνακες λειτουργούν ως iterables:

```
>>> for x in a:
...   print(x)
...
1
42
3
```

Προσέξτε όμως ότι σε πολυδιάστατους πίνακες, το iterable αυτό διατρέχει μόνο την πρώτη διάσταση:

```
>>> for x in b:
...   print(x)
...
[ 1 17  3]
[4 5 6]
```

δηλαδή τα δύο στοιχεία που τυπώθηκαν ήταν η πρώτη και η δεύτερη γραμμή αντίστοιχα. Μπορούμε να διατρέξουμε όλα τα στοιχεία ενός πολυδιάστατου πίνακα ως εξής:

```
>>> for x in b.flat:
...   print(x)
...
1
17
3
4
5
6
```

5. Οι πίνακες είναι **ομογενείς** δηλαδή όλα τους τα στοιχεία είναι του ίδιου τύπου. Αυτό είναι διαφορετικό από τη φιλοσοφία της Python, π.χ. οι λίστες της μπορούν να περιέχουν στοιχεία διαφορετικών τύπων. Το πεδίο `dtype` ενός

πίνακα μας δείχνει το data type των στοιχείων του.

```
>>> print(b.dtype)
int64
```

Το `np.int64` είναι ο τύπος των 64-bit ακέραιων αριθμών. Είναι διαφορετικός από τον τύπο των ακεραίων στην Python (`int`), ο οποίος υποστηρίζει αριθμούς οσοδήποτε μεγάλους (bignums). Εξίσου χρήσιμος, αν όχι χρησιμότερος, είναι ο τύπος `np.float64`.

```
>>> c = np.array([3.14, 2.78])
>>> print(c)
[3.14 2.78]
>>> print(c.dtype)
float64
```

6. Πίνακες με ιδιαίτερη μορφή και περιεχόμενο. Ο μηδενικός πίνακας:

```
>>> z = np.zeros((3, 3))
>>> print(z)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Προσέξτε ότι ο πίνακας αυτός έχει στοιχεία με τύπο `np.float64`. Αν θέλαμε να έχουμε ακέραια μηδενικά, μπορούμε να δώσουμε τιμή στην (προαιρετική) παράμετρο `dtype`:

```
>>> print(np.zeros((3, 3), dtype=np.int64))
[[0 0 0]
 [0 0 0]
 [0 0 0]]
```

Το ίδιο ακριβώς μπορούμε να πάρουμε με `dtype=int` — ο τύπος ακεραίων της Python αυτόματα μετατρέπεται σε `np.int64` σε αυτή την περίπτωση.

Ένας πίνακας γεμάτος με άσους, ή οποιοδήποτε άλλη σταθερά:

```
>>> n = np.ones((3, 3))
>>> print(n)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
>>> f = np.full((3, 3), 42)
>>> print(f)
[[42 42 42]
 [42 42 42]
 [42 42 42]]
```

Μοναδιαίος πίνακας:

```
>>> i = np.eye(3)
>>> print(i)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Πίνακες που αντιστοιχεί σε κάποιο `range`:

```
>>> print(np.arange(10))
[0 1 2 3 4 5 6 7 8 9]
>>> print(np.arange(4, 10))
[4 5 6 7 8 9]
>>> print(np.arange(4, 10, 2))
[4 6 8]
```

Πίνακας με τυχαίους αριθμούς, στο διάστημα $[0, 1)$.

```
>>> r = np.random.random((3, 3))
>>> print(r)
[[0.78302488 0.85854877 0.03570157]
```

```
 [0.70774711 0.24283631 0.92667939]
 [0.34008778 0.68366237 0.22990968]]
```

7. **Slicing**: Μία από τις χρησιμότερες λειτουργίες που υποστηρίζει το NumPy είναι η δυνατότητα να "κόβουμε φέτες" (slicing) πινάκων. Αυτό γίνεται με παρόμοια σύνταξή όπως αυτή που χρησιμοποιούμε π.χ. για φέτες λιστών στην Python

```
>>> s = i[:2, 1:3]
>>> print(s)
[[0. 0.]
 [1. 0.]]
>>> print(s[1, 0])
1.0
```

Εν αντιθέσει όμως με τις φέτες λιστών στην Python (που κάθε φέτα κατασκευάζει αντίγραφο των στοιχείων της αρχικής λίστας), στο NumPy οι φέτες είναι απλές "όψεις" (views) μέρους ενός πίνακα. Αυτό γίνεται αντιληπτό με το παρακάτω παράδειγμα, στο οποίο αλλάζουμε ένα στοιχείο της φέτας `s` και παρατηρούμε ότι αλλάζει και το αντίστοιχο στοιχείο του αρχικού πίνακα `i`:

```
>>> s[1, 0] = 42
>>> print(s)
[[ 0.  0.]
 [42.  0.]]
>>> print(i)
[[ 1.  0.  0.]
 [ 0. 42.  0.]
 [ 0.  0.  1.]]
```

Αν θέλουμε να αντιγράψουμε μια φέτα (η οποιονδήποτε άλλο πίνακα) μπορούμε να χρησιμοποιήσουμε τη μέθοδο `copy`:

```
>>> c = i[:2, 1:3].copy()
>>> c[1, 0] = 17
>>> print(c)
[[ 0.  0.]
 [17.  0.]]
>>> print(i)
[[ 1.  0.  0.]
 [ 0. 42.  0.]
 [ 0.  0.  1.]]
```

Οι φέτες μπορούν να περιορίζονται και σε συγκεκριμένα στοιχεία μίας διάστασης. Προσέξτε τη διαφορά ανάμεσα στα παρακάτω:

```
>>> print(i[2, 1:3])
[0. 1.]
>>> print(i[2:3, 1:3])
[[0. 1.]]
```

Το πρώτο είναι ένας μονοδιάστατος πίνακας, ενώ το δεύτερο ένας διδιάστατος πίνακας \(1 \times 2\).

8. **Mass indexing**: Μπορούν να χρησιμοποιηθούν πίνακες ως indices σε έναν πίνακα. Δείτε το παρακάτω παράδειγμα:

```
>>> i = np.eye(3)
>>> print(i)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> x = np.arange(3)
>>> print(x)
[0 1 2]
>>> y = np.array([2, 0, 1])
>>> print(y)
[2 0 1]
```

Χρησιμοποιώντας τα `x` και `y` ως indices στον πίνακα `i` παίρνουμε μία μονοδιάστατη όψη των στοιχείων `i[0,2]`, `i[1,0]` και `i[2,1]` (δηλαδή των στοιχείων `i[x[j], y[j]] for j in range(3)`).

```
>>> print(i[x, y])
[0. 0. 0.]
```

Η όψη αυτή είναι mutable:

```
>>> i[x, y] += 5
>>> print(i)
[[1. 0. 5.]
 [5. 1. 0.]
 [0. 5. 1.]]
```

9. **Boolean indexing**: Μπορεί επίσης να χρησιμοποιηθεί ένας πίνακας μέσα σε μία λογική παράσταση. Στην περίπτωση αυτή, τα στοιχεία του πίνακα συμμετέχουν ένα προς ένα στη λογική παράσταση και κατασκευάζεται ένας πίνακας με τις λογικές τιμές:

```
>>> j = i > 0
>>> print(j)
[[ True False False]
 [False  True False]
 [False False  True]]
```

Ο πίνακας `j` έχει ίδιες διαστάσεις με τον `i` και περιέχει `True` στις θέσεις όπου το αντίστοιχο στοιχείο του `i` είναι μεγαλύτερο του μηδέν.

Τέτοιοι πίνακες όπως ο `j` μπορούν επίσης να χρησιμοποιηθούν για mass indexing:

```
>>> print(i[j])
[1. 5. 5. 1. 5. 1.]
>>> print(i[i == 5])
[5. 5. 5.]
>>> i[i==5] = 42
>>> print(i)
[[ 1.  0. 42.]
 [42.  1.  0.]
 [ 0. 42.  1.]]
```

10. **Αριθμητική πινάκων**: Οι πίνακες μπορούν να χρησιμοποιηθούν σε αριθμητικές πράξεις και στην περίπτωση αυτή οι πράξεις εφαρμόζονται στα στοιχεία τους ένα προς ένα:

```
>>> print(x)
[0 1 2]
>>> print(y)
[2 0 1]
>>> print(x+y)
[2 1 3]
>>> print(x*y)
[0 0 2]
```

Προσέξτε ότι στην τελευταία εντολή υπολογίζεται το γινόμενο των επιμέρους στοιχείων των πινάκων `x` και `y`, που προφανώς είναι διαφορετικό τόσο από το γινόμενο πινάκων όσο και από το εσωτερικό γινόμενο. Αν θέλουμε το εσωτερικό γινόμενο των διανυσμάτων `x` και `y` μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `np.dot` ή τον τελεστή `@`:

```
>>> print(np.dot(x, y))
2
>>> print(x @ y)
2
```

Πολλές ακόμη συναρτήσεις ορίζονται στο NumPy, π.χ.

```
>>> print(np.sqrt(x))
[0.         1.         1.41421356]
```

Ο ανάστροφος ενός πίνακα κατασκευάζεται με τη συνάρτηση 'np.transpose' ή με την εξής συντομοτραφία:

```
>>> print(np.transpose(i))
[[ 1. 42.  0.]
 [ 0.  1. 42.]
```

```
 [42.  0.  1.]]
>>> print(i.T)
[[ 1. 42.  0.]
 [ 0.  1. 42.]
 [42.  0.  1.]]
```

Προσέξτε ότι και ο ανάστροφος είναι απλά μία όψη του πίνακα:

```
>>> i.T[0, 1] = 17
>>> i.T
array([[ 1., 17.,  0.],
       [ 0.,  1., 42.],
       [42.,  0.,  1.]])
>>> i
array([[ 1.,  0., 42.],
       [17.,  1.,  0.],
       [ 0., 42.,  1.]])
```

11. **Tiling**: Έστω ότι έχουμε ένα διδιάστατο πίνακα x και έναν μονοδιάστατο πίνακα v:

```
>>> x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
>>> print(x)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> v = np.array([1, 0, 1])
>>> print(v)
[1 0 1]
```

Θέλουμε να κατασκευάσουμε έναν πίνακα y προσθέτοντας τον πίνακα v σε κάθε γραμμή του x. Για να το κάνουμε αυτό, μπορούμε πρώτα να κατασκευάσουμε έναν "άδειο" πίνακα με τις ίδιες διαστάσεις όπως ο x και στη συνέχεια να αναθέσουμε το σωστό άθροισμα σε κάθε γραμμή του:

```
>>> y = np.empty_like(x)
>>> for i in range(4):
...     y[i] = x[i] + v
...
>>> print(y)
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Είναι όμως πολύ αποδοτικότερο, αντί να χρησιμοποιήσουμε ένα `for` loop, να "μεγαλώσουμε" τον πίνακα v αντιγράφοντάς τον σε τέσσερις γραμμές και κατασκευάζοντας έτσι ένα διδιάστατο πίνακα \(4 \times 3\):

```
>>> z = np.tile(v, (4, 1))
>>> print(z)
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

Στη συνέχεια, μπορούμε να προσθέσουμε τους πίνακες x και z και να πάρουμε το ίδιο αποτέλεσμα με πριν:

```
>>> y = x + z
>>> print(y)
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Ένα ακόμη παράδειγμα πλακόστρωσης (tiling), σε δύο διαστάσεις αυτή τη φορά:

```
>>> print(x)
[[ 1  2  3]
```

```
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> print(np.tile(x, (2, 3)))
[[ 1  2  3  1  2  3  1  2  3]
 [ 4  5  6  4  5  6  4  5  6]
 [ 7  8  9  7  8  9  7  8  9]
 [10 11 12 10 11 12 10 11 12]
 [ 1  2  3  1  2  3  1  2  3]
 [ 4  5  6  4  5  6  4  5  6]
 [ 7  8  9  7  8  9  7  8  9]
 [10 11 12 10 11 12 10 11 12]]
```

12. Το παραπάνω tiling του πίνακα v μπορεί να αποφευχθεί, χάρη σε ένα μηχανισμό που το NumPy ονομάζει **broadcasting**:

```
>>> y = x + v
>>> print(y)
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Το broadcasting καθορίζει πώς γίνονται οι πράξεις μεταξύ πινάκων με διαφορετικό σχήμα. Αυτό που συμβαίνει συνηθέστερα είναι ότι ο μικρότερος πίνακας "μεγαλώνει" για να φτάσει σε διαστάσεις τον μεγαλύτερο. Είναι η ίδια διαδικασία που ακολουθείται και όταν στις πράξεις συμμετέχουν αριθμοί αντί πινάκων:

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 2, 2])
>>> print(a*b)
[2 4 6]
>>> b = 2
>>> print(a*b)
[2 4 6]
```

Περισσότερα για το broadcasting και τις εφαρμογές του μπορείτε να βρείτε στην αντίστοιχη τεκμηρίωση του NumPy.

13. **Reshaping**: Μπορούμε να φτιάξουμε μία όψη διαφορετικού σχήματος με τη μέθοδο `reshape`:

```
>>> a = np.arange(15)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
>>> s = a.reshape(3, 5)
>>> print(s)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Προσέξτε όμως ότι και πάλι πρόκειται για όψη, όχι αντίγραφο:

```
>>> s[1, 1] = 42
>>> print(s)
[[ 0  1  2  3  4]
 [ 5 42  7  8  9]
 [10 11 12 13 14]]
>>> print(a)
[ 0  1  2  3  4  5 42  7  8  9 10 11 12 13 14]
```