ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μελέτη και υλοποίηση τεχνικών κατάστρωσης σχεδίου σε περιβάλλον Unity3D

Βασίλειος-Μάριος Σ. Αναστασίου
Παναγιώτης Χ. Διαμαντόπουλος

Επιβλέποντες: **Μανώλης Κουμπαράκης**, Καθηγητής
Σταύρος Βάσσος, Μεταδιδακτορικός Ερευνητής

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2011

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μελέτη και υλοποίηση τεχνικών κατάστρωσης σχεδίου σε περιβάλλον Unity3D

Βασίλειος-Μάριος Σ. Αναστασίου
Α.Μ.: 1115200600066

Παναγιώτης Χ. Διαμαντόπουλος
Α.Μ.: 1115200600028

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Μανώλης Κουμπαράκης**, Καθηγητής
Σταύρος Βάσσος, Μεταδιδακτορικός Ερευνητής

ΠΕΡΙΛΗΨΗ

Η παρούσα πτυχιακή εργασία αποσκοπεί στην υλοποίηση ενός κλασικού σχεδιαστή (classical planner) στη μηχανή βιντεοπαιχνιδιών Unity3D, με σκοπό να δημιουργηθεί ένας εύκολος, πρακτικός και γενικής χρήσης καταστρωτής σχεδίου. Θα μελετηθεί ο σχεδιασμός ακαδημαϊκού και ειδικού τύπου σχεδιαστών (όπως σε περιβάλλοντα παιχνιδιών) και θα αναλυθούν οι διαφορές τους, τις οποίες θα προσπαθήσουμε να εξομαλύνουμε στην υλοποίησή μας. Επιπλέον στόχος της πτυχιακής είναι να αποτελεί ιδανική εκπαιδευτική βάση για τη μελέτη και δημιουργία σχεδιαστών σε διαδραστικά περιβάλλοντα.

Στο πρώτο κομμάτι της παρούσας πτυχιακής εργασία θα παρουσιαστεί μια επισκόπηση του πεδίου της Τεχνητής Νοημοσύνης που αναφέρεται ως σχεδιασμός (planning). Θα αναφερθούμε σε σχεδιαστές ακαδημαϊκής χρήσης καθώς και σε σχεδιαστές ειδικού σκοπού που εφαρμόζονται σε περιβάλλοντα βιντεοπαιχνιδιών, συγκρίνοντας ταυτόχρονα τις δύο αυτές μεγάλες κατηγορίες.

Στο δεύτερο κομμάτι της εργασίας θα περιγραφεί το περιβάλλον Unity3D και ο υλοποιημένος από εμάς σχεδιαστής iThink. Θα αναλυθεί η χρήση του, η αρχιτεκτονική του και οι προκλήσεις υλοποίησης ενός σχεδιαστή σε πρακτικό περιβάλλον μηχανής παιχνιδιών, καθώς και τεχνικοί περιορισμοί/προβληματισμοί σε σχέση με έναν ακαδημαϊκό σχεδιαστή.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή Νοημοσύνη, Μηχανική Λογισμικού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: κατάστρωση σχεδίου, μηχανή παιχνιδιών, Unity3D, C#, STRIPS

ABSTRACT

On this thesis, we have created a classical planner on the Unity3D game engine. Our main purpose was to create a practical and easy to use domain-independent planner. Studies according to academic planning and special type planning (like the one that is used in video games) will be presented and their main differences will be analyzed. In our project, we try to normalize this kind of differences that arise between the two paradigms. An additional objective is to provide a library for educational purposes and an ideal starting ground for the study and development of planning software in interactive videogame environments.

The first half of the thesis will provide an overview of the academic discipline of Artificial Intelligence known as planning. The most common planners from the fields of academic research and game development will be described.

The second half of the thesis will present the Unity3D game engine and toolset, as well as our planning library “iThink”. We will fully describe the design choices made and the challenges presented during the development of the project, along with the practical limitations and issues involved.

SUBJECT AREA: Artificial Intelligence, Software Engineering

KEYWORDS: planning, game engine, Unity3D, C#, STRIPS

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

1. Εισαγωγή

1.1 Τεχνητή νοημοσύνη και αυτοματοποιημένος σχεδιασμός

Ο αυτοματοποιημένος σχεδιασμός (automated planning) είναι το πεδίο της Τεχνητής Νοημοσύνης που ασχολείται με την αναλυτική δημιουργία αλληλουχίας ενεργειών (πλάνο) που μπορούν να εκτελεστούν από έξυπνους/αναλυτικούς πράκτορες (Intelligent/reasoning agent) με σκοπό την επίτευξη ενός στόχου. Αυτό το πλάνο μπορεί να αποτελεί για παράδειγμα αλληλουχία κινήσεων ενός ρομποτικού βραχίονα, γενική στρατηγική πλοήγησης ενός αυτοματοποιημένου οχήματος, ακριβή αλληλουχία ενεργειών ενός έξυπνου πράκτορα, ή μια στρατηγική ανεφοδιασμού και διαμοιρασμού πόρων (logistics scheduling).

Το κεφάλαιο 2 της παρούσας πτυχιακής θα αφιερωθεί στην ανάλυση του αυτοματοποιημένου σχεδιασμού.

1.2 Η έννοια της τεχνητής νοημοσύνης σε βιντεοπαιχνίδια

1.2.1 «Προϊστορία»

Μέχρι πρόσφατα, η τεχνητή νοημοσύνη που χρησιμοποιούσαν τα παιχνίδια δε βασίζονταν σε τεχνικές που αναπτύχθηκαν από την ακαδημαϊκή κοινότητα. Οι λόγοι έχουν να κάνουν κυρίως με τα παρακάτω:

- αυτό που περιγραφόταν ως AI δεν ήταν πάντα μια υλοποιημένη μεθοδολογία αλλά απλά ένα σύνολο κανόνων που υλοποιούσε την εκτέλεση ενός παιχνιδιού,
- στις περιπτώσεις που χρησιμοποιούταν μια ακαδημαϊκή τεχνική, η υλοποίηση δεν αποσκοπούσε στην εύρεση βέλτιστης η "έξυπνης" λύσης αλλά στη γρήγορη εύρεση “αρκετά έξυπνης” λύσης, ούτως ώστε να προσδίδει την “αίσθηση νοημοσύνης” ενώ στην πραγματικότητα είναι τρικ για διασκεδαστικό παιχνίδι,
- σε παλιά συστήματα, η τεχνητή νοημοσύνη αφορούσε απλοϊκές συμπεριφορές πρακτόρων, όπως για παράδειγμα “εχθρούς” οι οποίοι κυνηγούσαν το χαρακτήρα του παίκτη, ή προ-προγραμματισμένες/παραμετρικές "αντιδράσεις" σε μεταβλητές όπως η θέση του παίχτη,
- μεγάλες απαιτήσεις σε χρόνο υλοποίησης και αποσφαλμάτωσης.

Στην ανάπτυξη βιντεοπαιχνιδιών, όταν αναφερόμασταν στη “Τεχνητή Νοημοσύνη” του βιντεοπαιχνιδιού δεν υπονοούσαμε απαραίτητα την ύπαρξη συστήματος έξυπνων πρακτόρων, μιας και το παιχνίδι μπορεί να υλοποιούσε απλά ένα σύνολο

τυποποιημένων λύσεων οι οποίες όριζαν τον τρόπο λειτουργίας των στοιχείων του παιχνιδιού.

1.2.2 Σύγχρονη ΤΝ σε βιντεοπαιχνίδια.

Τα τελευταία χρόνια οι εταιρίες, προσπαθώντας να ακολουθήσουν τις απαιτήσεις των παικτών για απαιτητικότερα παιχνίδια, έχουν αρχίσει να ξεφεύγουν από την τυποποιημένη συμπεριφορά των πρακτόρων και να προσπαθούν να βρουν τρόπους να εισάγουν μία πολυπλοκότητα στις αντιδράσεις και τους στόχους τους. Ταυτόχρονα τα ίδια τα παιχνίδια είναι πολύ ανώτερα σε χαρακτηριστικά αλλά και πολυπλοκότητα και απαιτούν αποδοτικές και έξυπνες υλοποιήσεις. Για παράδειγμα μπορεί να συμπεριλαμβάνουν στρατηγική οργάνωση, τακτικές προσαρμοσμένες στις ενέργειες του παίκτη, διαχείριση πολλών (π.χ. εκατοντάδων) πρακτόρων ταυτόχρονα, διαχείριση περιβαλλόντων A-life, δυναμική αναζήτηση μονοπατιών για μεμονωμένους αλλά και ομάδες πρακτόρων, και πολλές άλλες εφαρμογές.

Χρειάζονται λοιπόν αποδοτικά συστήματα, τα οποία κλιμακώνονται, παραμετροποιούνται, είναι εύκολα στη χρήση και παρέχουν μία τυχαιότητα στα αποτελέσματά τους για να παρέχουν τη δυνατότητα παραγωγής αληθοφανών και αναδυόμενων συμπεριφορών.

1.2.3 Οι ανάγκες της ΤΝ σε περιβάλλοντα βιντεοπαιχνιδιών

Υπάρχει πληθώρα αντιθέσεων στη χρήση ΤΝ (Τεχνητής Νοημοσύνης) σε ένα περιβάλλον βιντεοπαιχνιδιού σε σχέση με μια ακαδημαϊκή ή εμπορική εφαρμογή:

- Τα βιντεοπαιχνίδια είναι ψυχαγωγικό λογισμικό στο οποίο ο χρήστης (παίκτης) περιμένει να αντιμετωπίσει διαχειρίσιμες προκλήσεις ενώ ταυτόχρονα αναμένει μια εμπειρία από ψυχαγωγική μέχρι και ενθουσιώδη.
- Κατά τη διάρκεια της εκτέλεσης του βιντεοπαιχνιδιού λειτουργούν πολλά υποσυστήματα, όπως αυτά των γραφικών, που απαιτούν μεγάλο ποσοστό των διαθέσιμων πόρων του συστήματος. Είναι επίσης δυνατό να λειτουργεί ταυτόχρονα μεγάλο πλήθος πρακτόρων με ευρεία επιλογή λειτουργιών/στρατηγικών, το οποίο περιορίζει ακόμη περισσότερο το διαθέσιμο χρονικό προϋπολογισμό ανά πράκτορα.
- Η ΤΝ των παιχνιδιών είναι κατά κανόνα περιορισμένη και καλύπτει πολύ συγκεκριμένες ανάγκες σε αντίθεση με άλλα συστήματα γενικής χρήσης, καθώς επικεντρώνεται στη γρήγορη επεξεργασία μεγάλου πλήθους δεδομένων, ούτως ώστε να παράγει αποτελέσματα σε πραγματικό χρόνο.

Τα παραπάνω είχαν ως αποτέλεσμα το σύστημα της TN να είναι πολλές φορές εξειδικευμένο σε συγκεκριμένη εφαρμογή, ούτως ώστε να είναι πιο αποδοτική η λειτουργία του, και να είναι δύσκολη και χρονοβόρα η μεταφορά του σε άλλο περιβάλλον ή και εφαρμογή.

1.2.4 Γεφύρωση του χάσματος μεταξύ ακαδημαϊκού και εμπορικού κόσμου

Αρχικά οι μέθοδοι που μελετούσε σε θεωρητικό επίπεδο η ακαδημαϊκή κοινότητα για την εφαρμογή της τεχνητής νοημοσύνης δεν αποτελούσε στοιχείο αναφοράς και υλοποίησης σε ηλεκτρονικά παιχνίδια. Αυτό συνέβαινε κυρίως λόγο των διαφορετικών απαιτήσεων σε υπολογιστική ισχύ και ταχύτητα μεταξύ των μεθόδων που χρησιμοποιούνται σε βιντεοπαιχνίδια και των θεωρητικών μεθόδων. Τα τελευταία χρόνια όμως έχει αρχίσει να γεφυρώνεται το χάσμα ανάμεσα στην ακαδημαϊκή κοινότητα και τον εμπορικό κόσμο αφού ο δεύτερος έχει αρχίσει να αξιοποιεί τεχνικές από την πρώτη, οι οποίες είναι εύκολες στη χρήση και παρέχουν τις κατάλληλες λειτουργίες για την εξυπηρέτηση των σκοπών τους, ενώ η πρώτη δείχνει ενδιαφέρον για τις προκλήσεις υλοποίησης τεχνικών για τη λύση των τύπων προβλημάτων που εμφανίζονται στα βιντεοπαιχνίδια.

Χαρακτηριστικό παράδειγμα αποτελούν πλήθος εταιριών και συνεδρίων τα οποία οργανώνουν διαγωνισμούς για παιχνίδια τα οποία υλοποιούν και χρησιμοποιούν κάποια θεωρητική μέθοδο τεχνητής νοημοσύνης για τη διαχείριση των πρακτόρων και του περιβάλλοντος τους, ενθαρρύνοντας έτσι τους αναπτυκτές λογισμικού. Για παράδειγμα, ο διαγωνισμός General Game Playing (GGP)[1] αφορά στην ανάπτυξη συστημάτων που μπορούν να παίξουν παιχνίδια τα οποία δεν είναι γνωστά από πριν και οι κανόνες τους δίνονται πριν την έναρξη του παιχνιδιού. Ο διαγωνισμός λαμβάνει χώρα στα μεγαλύτερα διεθνή συνέδρια τεχνητής νοημοσύνης όπως αυτό που διοργανώνει ο σύνδεσμος Association for the Advancement of Artificial Intelligence (AAAI) και το International Joint Conference on Artificial Intelligence (IJCAI). Επίσης, το 2010[2] και 2011[3] στο συνέδριο AIIIDE (Artificial Intelligence and Interactive Digital Entertainment Conference) έλαβαν χώρα οι πρώτοι διαγωνισμοί AI Starcraft Competition οι οποίοι είναι άμεσα συνδεδεμένοι με το επιτυχημένο εμπορικό παιχνίδι

[1] http://games.stanford.edu/

[2] http://eis.ucsc.edu/StarCraftAICompetition

[3] http://skatgame.net/mburo/sc2011/

Starcraft. Ένα ακόμα παράδειγμα αποτελεί η εταιρία Google η οποία σε συνεργασία με το Πανεπιστήμιο του Waterloo διεξάγει κάθε δύο χρόνια ένα διαγωνισμό για τη δημιουργία πρακτόρων που θα συναγωνίζονται άλλους πράκτορες δεδομένου ενός παιχνιδιού.

1.3 Σχεδιασμός (planning) σε βιντεοπαιχνίδια

Γνωρίζοντας πλέον τις προϋποθέσεις που πρέπει να καλύπτει η ΤΝ ενός βιντεοπαιχνιδιού, αναγνωρίζουμε πως χρειάζονται αποδοτικές λύσεις που μειώνουν το χρόνο ανάπτυξης και αποσφαλμάτωσης των οντοτήτων του παιχνιδιού καθώς και των λειτουργιών που αυτές εκτελούν. Επίσης, η αληθοφανής λειτουργία τους και η ύπαρξη αναδυόμενης πολυπλοκότητας είναι ένα στοιχείο θετικό και επιθυμητό σε πολλές περιπτώσεις. Μια πολύ καλή εναλλακτική για να ικανοποιηθούν αυτές η ανάγκες είναι η χρήση μεθόδων σχεδιασμού, οι οποίες θα αναπτυχθούν στο κεφάλαιο 2.

1.4 Υλοποίηση της βιβλιοθήκης iThink στην πλατφόρμα Unity3D

Κατά τη μελέτη ενός νέου γνωστικού πεδίου και την υλοποίηση ενός προγραμματιστικού έργου με συγκεκριμένους στόχους, είναι επιθυμητό να αποφευχθούν όσο το δυνατόν περισσότερο χρονοβόρες διαδικασίες που δε συμβάλλουν άμεσα στην επίτευξη των στόχων. Ταυτόχρονα όμως, είναι επιθυμητή η μελέτη ενός εναλλακτικού τρόπου υλοποίησης, ώστε η προσπάθεια να περιέχει πρωτότυπα και χρήσιμα στοιχεία, ειδικά εάν ένας στόχος είναι η εκπαιδευτική αξία που μπορεί να προσφέρει το έργο.

Επιπλέον, από την αρχή της πτυχιακής μας ήταν επιθυμητό να υλοποιήσουμε μια βιβλιοθήκη που δε θα λειτουργούσε όπως ένας τυπικός ακαδημαϊκός σχεδιαστής, αλλά ταυτόχρονα θα περιείχε στοιχεία που θα τη καθιστούσαν εύκολη στη χρήση, επεκτάσιμη και με δυνατότητες να λειτουργεί επαρκώς αποδοτικά για τις ανάγκες ενός σύγχρονου βιντεοπαιχνιδιού.

Ως επιθυμητή λύση για να ικανοποιηθούν όλοι οι παραπάνω περιορισμοί επιλέχθηκε το περιβάλλον Unity3D, το οποίο αποτελεί ένα ενιαίο περιβάλλον μηχανής παιχνιδιών και εργαλείων ανάπτυξης. Θα είχαμε έτσι τη δυνατότητα να απαγκιστρωθούμε από τις ανάγκες δημιουργίας μιας πλατφόρμας πάνω στην οποία θα στηρίζαμε τη βιβλιοθήκη μας, ενώ θα είχαμε διαθέσιμα πολλά πλεονεκτήματα που θα προσέφερε η επιλογή αυτή. Το περιβάλλον και η βιβλιοθήκη θα αναλυθούν λεπτομερώς στο κεφάλαιο 3, και συγκεκριμένα στις υποενότητες 3.1 και 3.2.

2. ΘΕΩΡΗΤΙΚΗ ΜΕΛΕΤΗ

2.1 Εισαγωγή

Ως κατάστρωση σχεδίου (planning) ή σχεδιασμό περιγράφουμε τη διαδικασία εύρεσης αλληλουχίας ενεργειών (actions) που αν ακολουθηθούν διαδοχικά από μια αρχική κατάσταση (state) τότε επιτυγχάνουν ένα στόχο (goal), δηλαδή οδηγούμαστε σε μια κατάσταση στόχου και επιλύουμε ένα συγκεκριμένο πρόβλημα. Αυτό γίνεται με βάση τις γνώσεις μας για τον κόσμο - μέρος ή και το σύνολο αυτών αποτελούν μια κατάσταση - και τις διαθέσιμες ενέργειες, που επιφέρουν μεταβολές (γνώσης) στις καταστάσεις αυτές. Πολύ συχνά αυτή η αλληλουχία ενεργειών, αποκαλούμενη και ως σχέδιο (plan), εκτελείται από έναν έξυπνο πράκτορα (intelligent/rational agent), δηλαδή από μια αυτόνομη οντότητα τεχνητής νοημοσύνης, για την επίτευξη των στόχων του.

Τα προβλήματα τα οποία έχει τη δυνατότητα να επιλύσει η διαδικασία του σχεδιασμού είναι πολλά και αρκετές φορές διαφορετικά σε φύση και πολυπλοκότητα. Η ολοκλήρωση του πλάνου και η επίλυση των προβλημάτων μπορεί να απαιτούν διαφορετικό πλήθος χρονικών ή υπολογιστικών πόρων, ή ακόμη και πλήθος ή είδος πληροφοριών για τη περιγραφή τους, παρόλα αυτά υπάρχουν κάποια βασικά στοιχεία (αρχές) που έχουν γενική εφαρμογή. Τα στοιχεία αυτά παρουσιάζονται παρακάτω:

- **Καταστάσεις Αναζήτησης** (Search States): Κάθε πρόβλημα σχεδιασμού περιλαμβάνει ένα χώρο αναζήτησης (search space) ο οποίος αντιστοιχεί στο σύνολο των κόμβων αναζήτησης που μπορεί να προκύψουν κατά τη κατάστρωση σχεδίου. Κάθε ένας τέτοιος κόμβος αντιστοιχεί σε μία κατάσταση στην οποία μπορεί να φτάσει ο πράκτορας μετά από την εφαρμογή μιας συγκεκριμένης αλληλουχίας ενεργειών. Το σύνολο των καταστάσεων αναζήτησης μπορεί να είναι διακριτό (πεπερασμένο ή απείρως μετρίσιμο) ή συνεχές (απείρως μη μετρίσιμο).
Στις απαρχές του σχεδιασμού το σύνολο των κόμβων αναζήτησης αναφερόταν ρητά κατά την αναζήτηση. Στην πραγματικότητα, στα περισσότερα προβλήματα, το μέγεθος του χώρου αναζήτησης καταστάσεων είναι αρκετά μεγάλο, από άποψη πολυπλοκότητας και από άποψη συνόλου κόμβων αναζήτησης, οπότε δεν έχει ιδιαίτερο ενδιαφέρον η ρητή αναφορά του.

- **Ενέργειες** (Actions): Κάθε διαδικασία σχεδιασμού πρέπει να έχει ένα σύνολο από διαθέσιμες ενέργειες οι οποίες επιδρούν πάνω σε μία κατάσταση και τη μεταβάλλουν. Οι ενέργειες αυτές χρησιμοποιούνται από τη κατάστρωση σχεδίου για την ολοκλήρωση του πλάνου και την επίτευξη του ζητούμενου στόχου. Κάθε μία τέτοια ενέργεια διατηρεί

εσωτερικά όλες τις πληροφορίες και τις αλλαγές που πρέπει να γίνουν σε μία κατάσταση όταν η ενέργεια επιδράσει σε αυτήν. Είναι δυνατό αυτές οι ενέργειες να αλλάζουν κατά τη διάρκεια της κατάστρωσης σχεδίου, με κάποιες να μην είναι εφαρμόσιμες πάντα ενώ σε άλλες περιπτώσεις να γίνονται διαθέσιμες άλλες ενέργειες, το οποίο εξαρτάται από τον τύπο του προβλήματος και τον πράκτορα.

- **Αρχική Κατάσταση** (Initial/Start State) και **Κατάσταση Στόχου** (Goal State): Ένα πρόβλημα σχεδιασμού ξεκινάει από μία κατάσταση (αρχική κατάσταση) και προσπαθεί να φτάσει στην κατάσταση στόχου αξιοποιώντας κατάλληλα τις διαθέσιμες ενέργειές του.

- **Χρόνος** (Time): Ο χρόνος αποτελεί ακόμα ένα βασικό παράγοντα στην υλοποίηση της κατάστρωσης σχεδίου. Σε κάθε πρόβλημα σχεδιασμού, οι ενέργειες που πρέπει να γίνουν για την επίλυση του αποτελούν μία προκαθορισμένη αλληλουχία (είτε συγκεκριμένη είτε γενικότερη) απαραίτητη για την εκπλήρωση του στόχου. Σε αυτήν την περίπτωση ο χρόνος μας ενδιαφέρει με τη σχετική έννοια. Δηλαδή μας ενδιαφέρει να διατηρηθεί η σωστή αλληλουχία ενεργειών καθ' όλη τη διάρκεια της εκτέλεσης της κατάστρωσης σχεδίου. Εξίσου σημαντικός παράγοντας όμως είναι και η διάρκεια της διαδικασίας σχεδιασμού, όπως και ο χρόνος που απαιτείται για την πραγμάτωση του σχεδίου, τα οποία όμως μελετώνται ανά τύπο προβλήματος και πιθανότατα ανά περίπτωση, σύμφωνα με τις ανάγκες και τους περιορισμούς του χρήστη.

- Τέλος, το **κριτήριο εύρεσης πλάνου**: Το οποίο δηλώνει το επιθυμητό αποτέλεσμα του πλάνου που θα προκύψει αφού αξιοποιηθούν με τον επιθυμητό τρόπο οι ενέργειες και ο χώρος αναζήτησης καταστάσεων. Υπάρχουν δύο διαφορετικές προσεγγίσεις για την εύρεση ενός πλάνου με βάση το κριτήριο εύρεσης πλάνου.

1. Εύρεση **βέλτιστου** πλάνου: Η εύρεση ενός εφικτού πλάνου το οποίο να βελτιστοποιεί την απόδοση της μετάβασης από την αρχική κατάσταση, στην κατάσταση στόχου. Η έννοια της απόδοσης έχει πολλές πτυχές οι οποίες διαφέρουν σε κάθε πρόβλημα. Ως επί το πλείστον, βέλτιστα πλάνα, σε προβλήματα σχεδιασμού, θεωρούνται αυτά τα οποία ελαχιστοποιούν το συνολικό αριθμό ενεργειών που απαιτείται για τη μετάβαση στην κατάσταση στόχου ή απαιτούν το ελάχιστο χρονικό κόστος στην πραγματοποίησή τους.
2. Εύρεση **εφικτού** πλάνου: Άλλες φορές, δε μας ενδιαφέρει ή δεν είναι δυνατό να βρούμε το βέλτιστο πλάνο. Σε αυτές τις περιπτώσεις, μας ενδιαφέρει η εύρεση

οποιουδήποτε πλάνου για τη μετάβαση στη κατάσταση στόχου, ανεξάρτητα από την απόδοση του πλάνου.

Σχέδιο ή πλάνο (Plan) θεωρείται μία αλληλουχία ενεργειών η οποία οδηγεί σταδιακά από μία αρχική κατάσταση σε μία κατάσταση στόχου. Η κατασκευή του πλάνου, αν και είναι σταδιακή, δεν είναι πάντα εύκολη μιας και εξαρτάται από τη φύση των ενεργειών αλλά και από τη φύση των καταστάσεων στις οποίες μπορεί να βρεθεί η κατάστρωση σχεδίου.

Καταλήγοντας, πρέπει να αναφερθούμε στις σημαντικές διαφορές που έχει η κατάστρωση σχεδίου από τις συμβατικές μεθόδους αναζήτησης:

- καθιστά ευκολότερη την αποσύνθεση του προβλήματος (είναι δυνατό ο στόχος να χωριστεί σε ευκολότερα επιτεύξιμους υποστόχους),
- περιορίζει το εύρος της αναζήτησης (μπορούν να χρησιμοποιηθούν μόνο ενέργειες σχετικές με το πλάνο, ενώ η απλή αναζήτηση θα τις χρησιμοποιήσει όλες), και
- μας αποδεσμεύει από την εισαγωγή ειδικής για κάθε ξεχωριστό πρόβλημα ευρετικής συνάρτησης (αφού πλέον αρκεί για παράδειγμα να δούμε κατά πόσο διαφέρει η κατάσταση στόχου από τη τωρινή).

Σε οποιοδήποτε είδος σχεδιασμού χρησιμοποιούνται οι βασικές αρχές που μόλις περιγράψαμε. Παρά ταύτα, όπως και στους συμβατικούς αλγορίθμους αναζήτησης, υπάρχουν πολλοί τρόποι να υλοποιηθούν αυτές οι αρχές.

2.2 Χώρος αναζήτησης

Λαμβάνοντας ως αναφορά το χώρο αναζήτησης που χρησιμοποιεί η κατάστρωση σχεδίου για να πραγματοποιήσει την αναζήτηση, μπορούμε να κατατάξουμε τις γενικευμένου σκοπού διαδικασίες σχεδιασμού σε τέσσερεις μεγάλες κατηγορίες:

1. **Κλασική** (classical): Ο κλασικός σχεδιασμός είναι ο περισσότερο μελετημένος τύπος σχεδιασμού, με αλγορίθμους να αναπτύσσονται εδώ και περισσότερα από 40 χρόνια. Προϋποθέτουν πως το περιβάλλον είναι παρατηρήσιμο και ντετερμινιστικό (δεν είναι δυνατό η επίδραση μιας ενέργειας να έχει τυχαίο αποτέλεσμα) και πως οι ενέργειες είναι άμεσα εφαρμόσιμες. Χωρίζεται σε δύο μεγάλες υποκατηγορίες:

a. **Αναζήτηση στο χώρο καταστάσεων** (state-space search), όπου η κατάστρωση σχεδίου μεταβάλλει την αρχική κατάσταση και προσπαθεί να βρει μία αλληλουχία τροποποιήσεων-ενεργειών οι οποίες να συνδέουν την αρχική κατάσταση με την κατάσταση στόχου.

b. **Αναζήτηση στο χώρο σχεδίων** (plan-space search), όπου μεταβάλλεται το σχέδιο και ο σχεδιαστής προσπαθεί να το προσαρμόσει για να πληροί κάποιες προϋποθέσεις και να οδηγεί στην κατάσταση στόχου.

2. **Ιεραρχική** (hierarchical): Αυτή η διαδικασία είναι αρκετά πολύπλοκη στο σχεδιασμό, αλλά ευκολότερη στη διαισθητική κατανόηση. Η βασική της αρχή έγκειται στη διάσπαση των μεγάλων και πολύπλοκων διεργασιών του σχεδίου σε επιμέρους μικρότερες υπό-διεργασίες τις οποίες ο αλγόριθμος σχεδίασης καλείται να υλοποιήσει. Τελικός στόχος της διαδικασίας αυτής είναι η αποδόμηση όλων των επιμέρους διεργασιών στα μικρότερα δυνατά τους κομμάτια και η αντιστοίχιση τους με τις ενέργειες που επηρεάζουν και μεταβάλλουν την κατάσταση στην οποία βρίσκεται η κατάστρωση σχεδίου.

3. **Ικανοποιησιμότητας** (SAT): Η ιδέα σε αυτή τη μέθοδο είναι η κωδικοποίηση του προβλήματος σχεδιασμού σε ένα πρόβλημα το οποίο έχει ήδη λύση και υπάρχει ήδη ένας αποδοτικός αλγόριθμος που το επιλύει. Το πρόβλημα κωδικοποιείται σε ένα υπάρχον πρόβλημα ικανοποιησιμότητας με τη χρήση της προτασιακής λογικής. Στην πορεία εφαρμόζουμε τον αλγόριθμο λύσης του προβλήματος αυτού στο πρόβλημα σχεδιασμού προκειμένου να αποφασιστεί αν υπάρχει εφικτή και αποδοτική λύση. Σε αυτή τη διαδικασία μας διευκολύνει το γεγονός ότι πλέον υπάρχουν πολλοί αποδοτικοί αλγόριθμοι γενικού σκοπού για προβλήματα ικανοποιησιμότητας, στα οποία μπορούμε να αντιστοιχίσουμε πληθώρα προβλημάτων σχεδιασμού και επομένως να βρούμε αποδοτικά λύσεις.

4. **Διαζευκτική** (disjunctive): Η βασική ιδέα πίσω από αυτή τη κατηγορία είναι η δημιουργία μικρότερων πλάνων τα οποία περιέχουν ενέργειες οι οποίες χωρίζονται με λογική διάζευξης, ταξινομήσεις και βοηθητικούς περιορισμούς. Τέτοιου είδους πλάνα παρέχουν ένα μειωμένο χώρο αναζήτησης και το πλεονέκτημα ότι δεν παράγονται πολλές φορές κατά την κατάστρωση σχεδίου πλάνα που έχουν ήδη απορριφθεί. Αποτελεί ουσιαστικά μια γενίκευση των μεθόδων αναζήτησης στο χώρο σχεδίων και ικανοποιησιμότητας.

Σε επόμενη ενότητα θα αναπτυχθούν εκτενώς ο κλασσικός σχεδιασμός και τα ιεραρχικά δίκτυα.

2.3 Μέθοδοι σχεδιασμού

2.3.1 Κλασικός σχεδιασμός (Classical planning)

Ο κλασσικός σχεδιασμός περιλαμβάνει δύο από τις πέντε κατηγορίες που αναπτύχθηκαν στην ενότητα 2.2. Αυτές είναι, ο σχεδιασμός με αναζήτηση στο χώρο καταστάσεων (state space) και ο σχεδιασμός με αναζήτηση στο χώρο πλάνου (plan space). Θα ακολουθήσει σχετική ανάλυση στις ενότητες 2.3.2 μέχρι 2.3.4.

2.3.2 STRIPS: Μία μέθοδος αναπαράστασης προβλημάτων σχεδιασμού

Ένα πρόβλημα σχεδιασμού μπορεί πολλές φορές να περιγραφεί ως ένα σύνολο καταστάσεων, ενεργειών και στόχων. Οι αλγόριθμοι κατάστρωσης σχεδίου για κλασσικό σχεδιασμό για να είναι αποδοτικοί πρέπει να μπορούν να εκμεταλλευτούν τη λογική δομή του προγράμματος προκειμένου να την αξιοποιήσουν για τη δημιουργία πλάνου.

Έπρεπε λοιπόν να βρεθεί μία γλώσσα η οποία να είναι αρκετά παραστατική και να μπορεί να περιγράψει πληθώρα προβλημάτων σχεδιασμού διαφορετικής φύσης και πολυπλοκότητας αλλά ταυτόχρονα να περιέχει αρκετούς περιορισμούς για να μπορεί να χρησιμοποιηθεί από αποδοτικούς αλγόριθμους σχεδιασμού. Η γλώσσα που αναπτύχθηκε για το σκοπό αυτό και που χρησιμοποιείται και στην παρούσα πτυχιακή είναι η γλώσσα STRIPS [1] και τυποποιεί την:

- **αναπαράσταση των καταστάσεων**
- **αναπαράσταση του στόχου**
- **αναπαράσταση των ενεργειών**

Για την αναπαράσταση των καταστάσεων, η κατάστρωση σχεδίου χωρίζει το περιβάλλον (κόσμο) στο οποίο δρα σε λογικές συνθήκες που αναπαριστούν τη γνώση ως σύζευξη θετικών λεκτικών. Ένα σύνολο τέτοιων λεκτικών περιγράφει μία κατάσταση την οποία θα χρησιμοποιήσει η κατάστρωση σχεδίου για την παραγωγή πλάνου. Τα θετικά λεκτικά που περιγράφουν μία κατάσταση δεν πρέπει να περιέχουν συναρτήσεις δηλαδή λεκτικά με μεταβλητές χωρίς τιμή και πρέπει να είναι θεμελιώδη δηλαδή ένα λεκτικό να μην περιέχει άλλα λεκτικά. Πρέπει να τονιστεί πως ισχύει η υπόθεση του κλειστού κόσμου, ότι δηλαδή όλα τα λεκτικά που δεν αναφέρονται ρητά στην περιγραφή μιας κατάστασης θεωρούμε πως είναι ψεύδη.

Ο στόχος, όπως ακριβώς και μία κατάσταση, περιγράφεται μόνο από θετικά λεκτικά. Η βασική διαφορά ανάμεσα σε μια ενδιάμεση κατάσταση και στην κατάσταση στόχου είναι ότι η κατάσταση στόχου είναι μερικώς ορισμένη. Αυτό σημαίνει ότι μία κατάσταση

ικανοποιεί τη κατάσταση στόχου αν περιέχει όλα τα λεκτικά που υπάρχουν στην κατάσταση στόχου χωρίς να μας ενδιαφέρει αν περιέχει και άλλα λεκτικά.

Μια ενέργεια περιγράφεται από τρία βασικά στοιχεία. Το όνομά της μαζί με μία λίστα παραμέτρων που αντιστοιχεί στα αντικείμενα τα οποία χρησιμοποιούνται σε αυτή και όλα μαζί χρησιμεύουν για τον προσδιορισμό της, ένα σύνολο από προϋποθέσεις που πρέπει να ισχύουν για να μπορεί να εφαρμοστεί η ενέργεια και ένα σύνολο από επιδράσεις που επακολουθούν μετά την εκτέλεσή της.

Οι προϋποθέσεις αποτελούνται από τη σύζευξη θετικών λεκτικών, τα οποία δεν αποτελούν συναρτήσεις και δηλώνουν τι πρέπει να είναι αληθές σε μία κατάσταση προκειμένου να μπορεί η ενέργεια να εφαρμοστεί. Μεταβλητές επιτρέπεται να υπάρχουν αλλά αν υπάρχουν πρέπει να εμφανίζονται και στη λίστα παραμέτρων της ενέργειας.

Οι επιδράσεις είναι μία σύζευξη θετικών και αρνητικών λεκτικών (literals) τα οποία επίσης δεν αποτελούν συναρτήσεις. Οι επιδράσεις περιγράφουν τον τρόπο με τον οποίο θα μεταβληθεί μία κατάσταση όταν εφαρμοστεί η ενέργεια σε αυτή. Όταν ένα λεκτικό στις επιδράσεις είναι αληθές αυτό σημαίνει ότι είναι και αληθές στην κατάσταση που προκύπτει με την εφαρμογή της ενέργειας. Ενώ όταν ένα λεκτικό είναι ψευδές, σημαίνει ότι το λεκτικό αυτό είναι ψευδές στην κατάσταση που προκύπτει. Και εδώ επιτρέπεται να εμφανίζονται μεταβλητές αλλά όπως και στις προϋποθέσεις πρέπει να εμφανίζονται και στη λίστα παραμέτρων της ενέργειας.

Για να γίνει πιο κατανοητός ο τρόπος με τον οποίο μεταβάλλονται οι καταστάσεις καθώς και η χρήση των ενεργειών, χρησιμοποιούνται οι έννοιες της εφαρμόσιμης ενέργειας και του αποτελέσματος. Μία ενέργεια καλείται εφαρμόσιμη σε μία κατάσταση, όταν η δεδομένη κατάσταση περιέχει όλα τα θετικά λεκτικά που υπάρχουν στις προϋποθέσεις της ενέργειας. Μια κατάσταση καλείται αποτέλεσμα μιας άλλης, όταν σε αυτή έχουν εφαρμοστεί οι επιδράσεις μιας ενέργειας, δηλαδή όταν η νέα κατάσταση προκύπτει με βάση την αρχική αλλά περιέχει όλα τα θετικά λεκτικά που υπάρχουν στις επιδράσεις της ενέργειας και επιπλέον της έχουν αφαιρεθεί όλα τα λεκτικά που αντιστοιχούν στις αρνητικές επιδράσεις.

Πρέπει επίσης εδώ να σημειωθεί ότι ένα θετικό λεκτικό που υπάρχει ήδη σε μία κατάσταση δεν επαναπροστίθεται με τις επιδράσεις της ενέργειας, ενώ αντίστοιχα αν δεν υπάρχει ένα λεκτικό στην κατάσταση ενώ εμφανίζεται το αντίστοιχο αρνητικό στις επιδράσεις τότε απλώς παραβλέπεται. Τέλος, μπορούμε να ορίσουμε την έννοια της

λύσης σε ένα πρόβλημα σχεδιασμού που στην απλούστερη μορφή της μπορεί να θεωρηθεί ως ένα σύνολο ενεργειών οι οποίες όταν εκτελεστούν από την αρχική κατάσταση έχουν ως αποτέλεσμα μία κατάσταση στόχου.

Οι περισσότεροι αλγόριθμοι που έχουν σχεδιαστεί για κλασσικό σχεδιασμό και που θα εξετάσουμε παρακάτω εκμεταλλεύονται και χρησιμοποιούν τις υποθέσεις και ευκολίες της γλώσσας STRIPS με σκοπό την απλούστερη και αποδοτικότερη κατάστρωση σχεδίου.

2.3.3 Ο σχεδιασμός με αναζήτηση στο χώρο καταστάσεων

Όσον αφορά το σχεδιασμό με αναζήτηση στο χώρο καταστάσεων υπάρχουν αρκετές μέθοδοι υλοποίησής του. Οι βασικότερες είναι η εμπρόσθια ή προς τα εμπρός αναζήτηση (**Forward** search), η ανάστροφη ή προς τα πίσω αναζήτηση (**Backward** search) και η αμφικατευθυντήρια αναζήτηση (**Bidirectional** search). Επιπλέον, μπορούν να χρησιμοποιηθούν ευρετικές συναρτήσεις ούτως ώστε να βελτιωθεί/συγκεκριμενοποιηθεί η διαδικασία αναζήτησης (**Heuristics** search).

2.3.3.1 Προς-τα-εμπρός αναζήτηση

Η προς τα εμπρός αναζήτηση ξεκινάει από την αρχική κατάσταση και εφαρμόζοντας τις ενέργειες που έχει διαθέσιμες ο σχεδιαστής παράγει τις επόμενες διαδοχικές καταστάσεις. Η διαδικασία επαναλαμβάνεται μέχρις ότου βρεθεί το τελικό πλάνο ή εξαντληθούν όλες οι διαθέσιμες καταστάσεις που έχουν παραχθεί. Ένα από τα σημαντικότερα προβλήματα που έχει αυτή η μέθοδος και που αποτελεί χαρακτηριστικό για τις μεθόδους υλοποίησης αυτής της κατηγορίας είναι ο **παράγοντας διακλάδωσης** (branching factor). Ένας μεγάλος παράγοντας διακλάδωσης σε ένα πρόβλημα σχεδιασμού μπορεί να οδηγήσει στη κατανάλωση του χρόνου σε μελέτη ενεργειών μη σχετικών με τη δεδομένη κατάσταση.

```
FORWARD_SEARCH
1   Q.Insert(x_I) and mark x_I as visited
2   while Q not empty do
3       x ← Q.GetFirst()
4       if x ∈ X_G
5           return SUCCESS
6       forall u ∈ U(x)
7           x' ← f(x, u)
8           if x' not visited
9               Mark x' as visited
10              Q.Insert(x')
11          else
12              Resolve duplicate x'
13  return FAILURE
```

Εικόνα 1. Αλγόριθμος της Προς-τα-Εμπρός Αναζήτησης για Σχεδιασμό

2.3.3.2 Προς-τα-πίσω αναζήτηση

Η προς-τα-πίσω αναζήτηση ακολουθεί την αντίθετη διαδικασία από τη προς τα εμπρός αναζήτηση. Ξεκινώντας από την κατάσταση στόχου προσπαθεί να φτάσει στην αρχική κατάσταση εφαρμόζοντας κάθε φορά την αντίστροφη μετάβαση καταστάσεων (inverse state transition). Κάθε μία διαθέσιμη ενέργεια παράγει μια προηγούμενη κατάσταση και η διαδικασία επαναλαμβάνεται εώς ότου βρεθεί πλάνο που να επιλύει το πρόβλημα ή εξαντληθούν όλες οι διαθέσιμες καταστάσεις που έχουν παραχθεί. Οι ενέργειες πρέπει να πληρούν κάποιες προϋποθέσεις προκειμένου να μπορούν να εφαρμοστούν στις καταστάσεις και να παράγουν την προηγούμενη της. Αυτές οι προϋποθέσεις είναι:

1. Οι επιδράσεις (effects) της ενέργειας να μετατρέπουν σε θετικό τουλάχιστον ένα από τα γεγονότα (facts) που υπάρχουν στην κατάσταση στόχου.
2. Οι επιδράσεις της ενέργειας να μην αναιρούν με κάποιο τρόπο τα γεγονότα τα οποία απαρτίζουν την κατάσταση στόχο.

Ένα από τα σημαντικότερα προβλήματα της μεθόδου είναι το γεγονός πως μπορεί και αυτή να έχει αρκετά υψηλό παράγοντα διακλάδωσης με αποτέλεσμα να συμβαίνει το ίδιο φαινομένο που συμβαίνει και στην προς τα εμπρός αναζήτηση.

```
BACKWARD_SEARCH
1   Q.Insert(x_G) and mark x_G as visited
2   while Q not empty do
3       x' ← Q.GetFirst()
4       if x = x_I
5           return SUCCESS
6       forall u^-1 ∈ U^-1(x)
7           x ← f^-1(x', u^-1)
8           if x not visited
9               Mark x as visited
10              Q.Insert(x)
11          else
12              Resolve duplicate x
13  return FAILURE
```

Εικόνα 2. Αλγόριθμος της Προς-τα-Πίσω Αναζήτησης για Σχεδιασμό

Υπάρχει τρόπος στην προς-τα-πίσω αναζήτηση να ελαττώσουμε τον παράγοντα διακλάδωσης. Αυτό μπορεί να επιτευχθεί αν αρχικοποιήσουμε μερικώς τους τελεστές οι οποίοι μας καθορίζουν τα βήματα για την προηγούμενη κατάσταση. Η βασική ιδέα πίσω

από αυτή την τεχνική (**Lifting**) είναι να καθυστερήσει την αντιστοίχιση αγνώστων μεταβλητών σε τιμές, μέχρι αυτές να χρειαστούν για την ικανοποίηση μιας κατάστασης στόχου ή υπό-στόχου. Αυτού του είδους η αναζήτηση έχει μικρότερο παράγοντα διακλάδωσης από την προς-τα-πίσω αναζήτηση αλλά είναι πιο πολύπλοκη στην υλοποίηση.

Ακόμα και με τη χρήση της ανύψωσης, ο χώρος αναζήτησης της προς-τα-πίσω αναζήτησης μπορεί να είναι ιδιαίτερα μεγάλος. Ας υποθέσουμε ότι έχουμε μία ενέργεια d και τις ενέργειες a, b, c οι οποίες είναι ανεξάρτητες μεταξύ τους και πως γνωρίζουμε ότι η ενέργεια d προηγείται των a, b, c, αλλά δεν υπάρχει κάποιο μονοπάτι που να οδηγεί από την αρχική κατάσταση στο d. Με τη χρήση της προς-τα-πίσω αναζήτησης θα αναπτύξουμε όλους τους δυνατούς συνδυασμούς μεταξύ των a, b, c πριν αντιληφθούμε ότι το πρόβλημά μας δεν έχει λύση.

Προκειμένου να μπορέσουμε να εξάγουμε πιο γρήγορα συμπέρασμα, πρέπει να βρούμε ένα τρόπο να μειώσουμε ακόμα περισσότερο το χώρο αναζήτησης αποκλείοντας νωρίς περιπτώσεις που δεν οδηγούν σε επιθυμητό αποτέλεσμα. Για να το πετύχουμε αυτό χρησιμοποιούμε την αναζήτηση με τη χρήση ευρετικών συναρτήσεων (**Heuristic** search).

2.3.3.3 Ευρετική αναζήτηση

Η αναζήτηση με τη χρήση ευρετικών συναρτήσεων αντιμετωπίζει το πρόβλημα σχεδιασμού ως ένα αφηρημένο πρόβλημα αναζήτησης που κάθε βήμα του περιλαμβάνει μεταβολή του πλάνου ενεργειών, διακλάδωσης και κλαδέματος του χώρου αναζήτησης. Προκειμένου να υλοποιήσουμε μία ντετερμινιστική αναζήτηση πρέπει να υπάρχει μία συνάρτηση η οποία να μας επιλέγει μία κατάσταση προς επεξεργασία από το σύνολο καταστάσεων.

Με την έννοια «κόμβο αναζήτησης» εννοούμε ένα ζευγάρι (A,U) όπου A είναι ένα μερικώς ορισμένο πλάνο με ένα σύνολο ενεργειών και U μία κατάσταση η οποία είναι η κατάσταση στην οποία θα καταλήξουμε αν από την αρχική μας ακολουθήσουμε το μερικώς ορισμένο πλάνο.

«Ευρετική συνάρτηση» για την επιλογή κόμβων αναζήτησης καλείται οποιοσδήποτε τρόπος μπορεί να μας παρέχει μία εκτίμηση για τη σχετικότητα του κόμβου που εξετάζουμε σε σχέση με την κατάσταση στόχου. Με άλλα λόγια, η εκτίμηση δηλώνει πόσο κοντά θεωρούμε πως είμαστε στην εύρεση κατάστασης-στόχου. Στη συνέχεια θα εξετάσουμε τις βασικές αρχές για τη δημιουργία τέτοιων συναρτήσεων καθώς και τις

συναρτήσεις που χρησιμοποιούνται στο σχεδιασμό με αναζήτηση στο χώρο καταστάσεων.

Οι ευρετικές συναρτήσεις χρησιμοποιούνται για να επιλύουμε μη ντετερμινιστικές επιλογές, αφού δεν υπάρχει ντετερμινιστικός τρόπος να αποφασίσουμε το σωστό κόμβο αναζήτησης για να συνεχίσουμε την αναζήτησή μας. Η ευρετική συνάρτηση δεν είναι αλάνθαστη, δηλαδή δεν σημαίνει ότι ο κόμβος που μας προτείνει για την αναζήτηση είναι πάντα η καλύτερη επιλογή ή ότι οδηγεί πάντα σε βέλτιστη ή ακόμα και πιο κοντά σε λύση. Επιθυμούμε όμως μία ευρετική συνάρτηση να κάνει το μικρότερο δυνατό αριθμό λαθών κατά τη διαδικασία της αναζήτησης καθώς και να είναι εύκολα υπολογίσιμη για να είναι ξεκάθαρο το πλεονέκτημα χρήσης της έναντι των τυχαίων επιλογών ή της εξαντλητικής αναζήτησης, που εξετάζει όλες τις εναλλακτικές επιλογές.

Οι ευρετικές συναρτήσεις επιλογής κόμβου στηρίζονται συνήθως στην αρχή της χαλαρότητας (relaxation principle). Προκειμένου να αξιολογήσουν πόσο «επιθυμητός» είναι ένας κόμβος για την αναζήτηση η ευρετική συνάρτηση επιλύει ένα χαλαρό πρόβλημα το οποίο προκύπτει από το βασικό, απλοποιώντας τις υποθέσεις του και χαλαρώνοντας τους περιορισμούς του. Η εκτίμηση επιτυγχάνεται χρησιμοποιώντας τον κόμβο προς αναζήτηση για να επιλύσει αυτό το χαλαρό πρόβλημα. Η λύση που προκύπτει από την επίλυση του χαλαρού προβλήματος χρησιμοποιείται ως εκτίμηση της λύσης που θα προκύψει αν χρησιμοποιήσουμε τον κόμβο αναζήτησης για να λύσουμε το πραγματικό πρόβλημα σχεδιασμού.

Όσο πιο κοντά στο πραγματικό πρόβλημα είναι το χαλαρό πρόβλημα που θα δημιουργηθεί τόσο πιο συνεπής θα είναι η εκτίμηση που θα προκύψει. Από την άλλη πλευρά, όσο πιο απλό είναι το χαλαρό πρόβλημα που δημιουργηθεί τόσο πιο εύκολα θα είναι να υπολογιστεί η ευρετική του. Εδώ λοιπόν εντοπίζεται η δυσκολία που υπάρχει στην υλοποίηση μιας ευρετικής συνάρτησης, αφού είναι δεν είναι εύκολο να βρεθεί η χρυσή τομή ανάμεσα στη συνέπεια και στην υπολογιστική ευκολία αυτής. Για να επιτευχθεί αυτό είναι αναγκαία η γνώση της δομής του προβλήματος πληροφορία που δεν είναι πάντα γνωστή ή προσβάσιμη.

Υπάρχει ένα βασικό χαρακτηριστικό το οποίο πρέπει να έχει μία ευρετική συνάρτηση για να θεωρηθεί καλή και να μπορεί να αξιοποιηθεί σωστά από τους αλγορίθμους που θα μελετήσουμε παρακάτω: Δεν πρέπει να υπερεκτιμά ποτέ το κόστος ενός κόμβου αναζήτησης, δηλαδή θα πρέπει σε κάθε περίπτωση να επιστρέφει το πολύ την ακριβή εκτίμηση που αντιστοιχεί σε μία δεδομένη κατάσταση. Μία τέτοια συνάρτηση ονομάζεται παραδεκτή και αποτελεί ένα καλό εργαλείο για το σχεδιασμό με τη χρήση ευρετικών

συναρτήσεων, αφού είναι σίγουρο πως θα οδηγήσει σε βέλτιστη λύση. Μη παραδεκτές συναρτήσεις μπορεί να οδηγήσουν ταχύτερα σε λύση, αλλά είναι δυνατό να οδηγήσουν σε μη-βέλτιστη λύση.

Υπάρχουν αρκετοί αλγόριθμοι σχεδιασμού που χρησιμοποιούν ευρετικές συναρτήσεις για την καθοδήγηση της αναζήτησης. Εδώ όμως θα μελετηθούν δύο βασικοί αλγόριθμοι που χρησιμοποιούνται στον κλασσικό σχεδιασμό. Ο πρώτος αλγόριθμος είναι η αναζήτηση πρώτα-στο-καλύτερο (Best First Search) ενώ ο δεύτερος αλγόριθμος είναι ο A* (A star).

2.3.3.4 Πρώτα-στο-καλύτερο αναζήτηση

Ο αλγόριθμος αναζήτησης πρώτα-στο-καλύτερο θεωρείται ο πρώτος αλγόριθμος της κατηγορίας των αλγορίθμων που χρησιμοποιούν ευρετικές συναρτήσεις και επομένως έχουν επιπλέον γνώση για το χώρο αναζήτησης (informed search algorithms).

Η βασική ιδέα πίσω από αυτόν τον αλγόριθμο είναι η χρήση μιας ευρετικής συνάρτησης για την εκτίμηση της σημαντικότητας κάθε κατάστασης (κόστος μετάβασης στην κατάσταση) που παράγεται ενώ το πλάνο εξελίσσεται. Σε κάθε επανάληψη κάθε μία καινούρια κατάσταση που δημιουργείται από την εφαρμογή των ενεργειών εκτιμάται (υπολογίζεται το κόστος της) και στη συνέχεια το σύνολο των καινούριων καταστάσεων που έχουν δημιουργηθεί μαζί με τις ήδη υπάρχουσες καταστάσεις ταξινομείται σε αύξουσα σειρά κόστους. Η διαδικασία επαναλαμβάνεται εξετάζοντας και επεκτείνοντας την κατάσταση με το ελάχιστο.

Ο αλγόριθμος αυτός έχει δεν βρίσκει τη βέλτιστη λύση αφού δεν διατηρεί το συνολικό κόστος που έχει καλύψει, αλλά εξετάζει σε κάθε επανάληψη μόνο το τρέχον κόστος. Αυτό το γεγονός μπορεί να δημιουργήσει πολλά περισσότερα προβλήματα αφού ο αλγόριθμος είναι δυνατό να βρεθεί σε ατέρμονα βρόχο και να επεκτείνει ένα απεριόριστο μονοπάτι αν η ευρετική συνάρτηση αξιολογεί κάθε μία κατάσταση σε ένα τέτοιο μονοπάτι ως τη βέλτιστη επιλογή. Παρά ταύτα, με τη χρήση μιας καλής ευρετικής συνάρτησης στα περισσότερα προβλήματα σχεδιασμού αυτά τα προβλήματα αντιμετωπίζονται.

2.3.3.5 A*

Ένας άλλος αλγόριθμος ο οποίος λαμβάνει υπόψη του και το κόστος που έχει διανύσει μέχρι τώρα και που είναι ιδιαίτερα αποτελεσματικός σε προβλήματα σχεδιασμού είναι ο αλγόριθμος A*.

Η ευρετική του συνάρτηση είναι ένα άθροισμα της μορφής f(n) = g(n)+h(n) όπου

- g(n) είναι το κόστος που χρειαστήκαμε μέχρι να φτάσουμε στην κατάσταση n
- h(n) είναι το εκτιμώμενο κόστος που πρέπει να διανύσει ο αλγόριθμος για να φτάσει στην κατάσταση στόχου
- f(n) είναι το συνολικό κόστος που χρειάζεται ο αλγόριθμος για να φτάσει από την αρχική κατάσταση στην κατάσταση στόχου μέσω της κατάστασης n.

Έχει αποδειχθεί ότι όταν η ευρετική συνάρτηση που χρησιμοποιεί ο αλγόριθμος είναι παραδεκτή δηλαδή δεν υπερεκτιμά, ο A* θα βρίσκει πάντα το μονοπάτι με το μικρότερο κόστος (βέλτιστο δυνατό) από μία αρχική κατάσταση σε μία κατάσταση στόχου.

Ο A* είναι ο κατεξοχήν αλγόριθμος αναζήτησης που χρησιμοποιείται σε πάρα πολλές εφαρμογές που χρησιμοποιούν κλασσικό σχεδιασμό λόγω της ιδιότητάς του να παράγει αποδοτικά βέλτιστες λύσεις. Ιδιαίτερη είναι η χρησιμότητά του στα ηλεκτρονικά παιχνίδια όπου χρησιμοποιείται για το σχεδιασμό κινήσεων και την αναζήτηση μονοπατιών (pathfinding). Στο ηλεκτρονικό παιχνίδι F.E.A.R. χρησιμοποιήθηκε εκτενώς για την πλήρη περιγραφή των πρακτόρων του παιχνιδιού, τόσο στο επίπεδο του σχεδιασμού όσο και στην αναζήτηση μονοπατιών [2].

Σε αυτήν την ενότητα αναφέραμε το σχεδιασμό ως μία αναζήτηση για ένα μονοπάτι από μία αρχική κατάσταση σε μία κατάσταση στόχο. Ο χώρος αναζήτησής μας ήταν οι καταστάσεις, και οι μεταβάσεις από μία κατάσταση σε μία άλλη μας δινόταν από τις κατάλληλες ενέργειες. Η διαδικασία καλείται σχεδιασμός (ή κατάστρωση σχεδίου) με αναζήτηση στο χώρο καταστάσεων. Στην επόμενη ενότητα θα μελετήσουμε μία ακόμα κατηγορία του κλασσικού σχεδιασμού, το σχεδιασμό με αναζήτηση στο χώρο πλάνων.

2.3.4 Ο σχεδιασμός με αναζήτηση στο χώρο πλάνων

Στο σχεδιασμό με αναζήτηση στο χώρο πλάνων ο χώρος αναζήτησης είναι ένα σύνολο από μερικώς ορισμένα πλάνα. Κάθε ενέργεια μεταβολής του πλάνου (plan refinement operation) που πραγματοποιείται έχει ως στόχο την περαιτέρω ολοκλήρωση του μερικώς ορισμένου πλάνου. Δηλαδή να επιτύχει έναν στόχο που δεν έχει ικανοποιηθεί (**ανοιχτό στόχο**) αναγκαίο για την ολοκλήρωση του πλάνου ή να επιλύσει τυχόν ασυνέπειες που παρουσιάζονται. Αυτού του είδους ο σχεδιασμός ακολουθεί την αρχή της μικρότερης δέσμευσης (least commitment principle) δηλαδή ο αλγόριθμος δεν εισάγει κανένα περιορισμό στο πλάνο εκτός και αν είναι απολύτως απαραίτητος για τη μεταβολή του. Στη συνέχεια, θα μελετήσουμε το χώρο αναζήτησης της μεθόδου.

Κάθε ένας κόμβος στο χώρο αναζήτησης μπορεί να περιγραφεί ως ένα μερικώς ορισμένο πλάνο. Κάθε ένα τέτοιο πλάνο περιλαμβάνει δύο στοιχεία που το περιγράφουν. Το πρώτο είναι ένα σύνολο από μερικώς αρχικοποιημένες ενέργειες ενώ το δεύτερο στοιχείο είναι ένα σύνολο από περιορισμούς πάνω σε αυτές τις ενέργειες.

Αναλυτικότερα κάθε μία ενέργεια εισάγεται προκειμένου να κάνει πιο συγκεκριμένο το πλάνο και εντέλει να εξαχθεί το πλάνο λύσης του προβλήματος. Με τη εισαγωγή της ενέργειας, εισάγονται και οι αντίστοιχοι περιορισμοί οι οποίοι χωρίζονται σε τρεις μεγάλες ομάδες. Οι περιορισμοί σειράς (Ordering constraints) που καθορίζουν τη σειρά με την οποία πρέπει να πραγματοποιηθούν οι ενέργειες. Οι περιορισμοί αυτοί ακολουθούν την αρχή της μικρότερης δέσμευσης, δηλαδή στο πλάνο θα εισαχθούν περιορισμοί μόνο όταν αυτοί είναι απολύτως απαραίτητοι. Μετά είναι οι αιτιολογικοί σύνδεσμοι (causal links) που καθορίζουν με τη σειρά τους τη σχέση ανάμεσα σε δύο ενέργειες.

Οι αιτιολογικοί σύνδεσμοι καθορίζουν αν μία από τις προϋποθέσεις μίας ενέργειας υποστηρίζεται και χρειάζεται από μία άλλη ενέργεια. Στην ουσία οι αιτιολογικοί σύνδεσμοι καθιστούν αναγκαία της ύπαρξης μιας ενέργειας στο πλάνο για την εκπλήρωση ενός ανοιχτού στόχου. Αν δεν υπάρχει αιτιολογικός σύνδεσμος για μία προϋπόθεση αυτή θεωρείται ως ένας ανοιχτός στόχος που πρέπει να λυθεί για να επιτύχει το πλάνο. Πρέπει εδώ να σημειωθεί ότι κάθε τέτοιος σύνδεσμος συνοδεύεται από ένα περιορισμό σειράς, αλλά δεν ισχύει το αντίθετο. Ακόμα πρέπει να αναφερθεί ότι ο αιτιολογικός σύνδεσμος δεν υποχρεώνει τις δύο ενέργειες να είναι σε σειρά. Αυτό σημαίνει ότι ανάμεσα σε δύο ενέργειες που συνδέονται με ένα τέτοιο σύνδεσμο, μπορούν να παρεμβάλλονται και άλλες ενέργειες που να χρησιμοποιούνται για διαφορετικές λειτουργίες στο πλάνο.

Τέλος, υπάρχουν οι περιορισμοί δέσμευσης μεταβλητών (variable binding constraints) που αντιστοιχίζουν μία μεταβλητή που έχει εισαχθεί στο πλάνο με μία συγκεκριμένη τιμή ή με μία άλλη μεταβλητή (substitutions, unifications), ή ρυθμίζουν θέματα ανισοτήτων μεταξύ των μεταβλητών του πλάνου (ισότητες, ανισότητες).

Θα μελετήσουμε δύο αλγόριθμούς υλοποίησης σχεδιασμού με αναζήτηση στο χώρο πλάνων. Ο πρώτος αλγόριθμος είναι η διαδικασία **PSP** και ο δεύτερος αλγόριθμος η διαδικασία **POP**.

Για να γίνουν οι διαδικασίες πιο κατανοητές θα πρέπει πρώτα να δώσουμε μερικούς ορισμούς οι οποίοι χρησιμοποιούνται στους αλγορίθμους με αναζήτηση στο χώρο πλάνων. Αυτοί είναι οι ορισμοί της **απειλής** (threat), του **σφάλματος** (flaw). Μία ενέργεια αποτελεί απειλή για ένα αιτιολογικό σύνδεσμο όταν i) έχει μία αρνητική επίδραση η οποία είναι ασύμβατη με αυτόν (δηλαδή μπορεί να το διαγράψει), ii) οι περιορισμοί σειράς για τις δύο ενέργειες που συνδέονται με το αιτιολογικό σύνδεσμο είναι ασυνεπείς με το σύνολο των περιορισμών σειράς που έχουν τεθεί, και iii) οι περιορισμοί δέσμευσης μεταβλητών για την ενοποίηση της επίδρασης της ενέργειας και της προϋπόθεσης που στηρίζει τον αιτιολογικό σύνδεσμο είναι ασυνεπείς με το σύνολο των περιορισμών δέσμευσης μεταβλητών που έχουν εισαχθεί. Σφάλμα σε ένα πλάνο καλείται είτε ένας ανοιχτός υπό-στόχος είτε μία απειλή.

Η διαδικασία **PSP** (Plan Space Planning) έχει ένα σύνολο από μερικώς ορισμένα πλάνα τα οποία της παρέχουν τα απαραίτητα στοιχεία για την υλοποίηση της κατάστρωσης σχεδίου. Ένα πλάνο θεωρείται λύση όταν δεν περιέχει σφάλματα. Άρα βασικός στόχος της διαδικασίας αυτής είναι να μεταβάλει ένα πλάνο για να εξαλείψει τα σφάλματά του, και να διατηρήσει συνεπή τα σύνολα των περιορισμών σειράς και των περιορισμών δέσμευσης μεταβλητών που έχουν εισαχθεί. Η διαδικασία επίλυσης των ψεγαδιών ακολουθεί κάποια βήματα:

1. Εύρεση των ψεγαδιών που περιέχει ένα πλάνο
2. Επιλογή ενός από αυτά
3. Εύρεση τρόπου επίλυσής του
4. Επιλογή μιας μεθόδου επίλυσής του
5. Μεταβολή του πλάνου σύμφωνα με τη μέθοδο που επιλέχθηκε

Αν ένα πλάνο δεν περιέχει καθόλου σφάλματα, τότε είναι πλάνο λύσης του προβλήματος σχεδιασμού. Αν πάλι ένα πλάνο περιέχει ένα σφάλμα το οποίο δεν μπορεί να λυθεί τότε το πλάνο δεν μπορεί να αναχθεί σε πλάνο λύσης

Μπορούμε με βάση αυτά τα βήματα να δημιουργήσουμε ένα αναδρομικό αλγόριθμο της διαδικασίας:

PSP(π)
  $flaws \leftarrow$ OpenGoals(π) $\cup$ Threats(π)
  if $flaws = \emptyset$ then return(π)
  select any flaw $\phi \in flaws$
  $resolvers \leftarrow$ Resolve(ϕ, π)
  if $resolvers = \emptyset$ then return(failure)
  nondeterministically choose a resolver $\rho \in resolvers$
  $\pi' \leftarrow$ Refine(ρ, π)
  return(PSP(π'))
end

Εικόνα 3. Αλγόριθμος διαδικασίας PSP

Στον αλγόριθμο υπάρχουν οι παρακάτω ρουτίνες που αποτελούν το σύνολο των βημάτων που αναφέρθηκαν. Οι συναρτήσεις OpenGoals() και Threats() που επιστρέφουν όλα τα σφάλματα που περιέχει ένα πλάνο. Η συνάρτηση Resolve() δεδομένου ενός ψεγαδιού βρίσκει όλους τους τρόπους επίλυσής του. Η διαδικασία αυτή χωρίζεται σε δύο βήματα.

- Αν το σφάλμα είναι ένας υποστόχος για μία προϋπόθεση μιας ενέργειας τότε έχει δύο τρόπους επίλυσης. Ένας είναι η δημιουργία νέου αιτιολογικού συνδέσμου με μία υπάρχουσα ενέργεια, η οποία περιέχει επίδραση που μπορεί να εισάγει τη ζητούμενη προϋπόθεση στο πλάνο για την επίλυση του υποστόχου, εισάγοντας πάντα και τους αντίστοιχους περιορισμούς σειράς και δέσμευσης μεταβλητών. Ο άλλος είναι η εισαγωγή μιας νέας ενέργειας στο πλάνο η οποία μπορεί να εισάγει την επιθυμητή προϋπόθεση για την επίλυση του στόχου. Σε αυτή την περίπτωση δημιουργείται ένας καινούριος αιτιολογικός σύνδεσμος μεταξύ των δύο ενεργειών και εισάγονται και οι απαραίτητοι αιτιολογικοί σύνδεσμοι και περιορισμοί δέσμευσης μεταβλητών.

- Αν το σφάλμα είναι μία απειλή σε έναν αιτιολογικό σύνδεσμο, τότε υπάρχουν τρεις τρόποι επίλυσης. Ο πρώτος τρόπος είναι να εισάγουμε έναν περιορισμό σειράς πριν από αυτόν, αν είναι συνεπής με το σύνολο των περιορισμών σειράς που έχουν

εισαχθεί. Ο δεύτερος τρόπος είναι να εισάγουμε έναν περιορισμό σειράς μετά τον αιτιολογικό σύνδεσμο, εφόσον είναι συνεπής με το σύνολο των περιορισμών σειράς που έχουν εισαχθεί. Τέλος, ο τρίτος τρόπος είναι να εισαχθεί ένας περιορισμός δέσμευσης μεταβλητών που να καθιστά την προϋπόθεση και την επίδραση μη ενοποιήσιμες.

Καταλήγοντας, υπάρχει η συνάρτηση Refine() η οποία μεταβάλλει το πλάνο με βάση τα αποτελέσματα της συνάρτησης Resolve(). Η διαδικασία αυτή δεν περιέχει ελέγχους και δεν κάνει οπισθοδρομήσεις αφού όλοι οι απαραίτητοι έλεγχοι για τη συνέπεια των περιορισμών και του πλάνου έχουν γίνει στη συνάρτηση Resolve().

Πρέπει και εδώ να τονιστεί ότι η χρήση ευρετικής συνάρτησης είναι απαραίτητη για την καθοδήγηση της αναζήτησης.

Η διαδικασία που περιγράψαμε (PSP procedure) αποτελεί ένα γενικευμένο αλγόριθμο για το σχεδιασμό με αναζήτηση στο χώρο πλάνων. Υπάρχουν αρκετές διαφορετικές υλοποιήσεις αυτού του αλγορίθμου. Εδώ θα περιγραφεί εκτενώς η διαδικασία **POP** (partial order planning).

Η βασική διαφορά ανάμεσα στις δύο διαδικασίες είναι στον τρόπο που διαχειρίζονται τις απειλές και τους υποστόχους. Ενώ η διαδικασία PSP χειρίζεται τις απειλές και τους υποστόχους με τον ίδιο τρόπο, επιλέγοντας σε κάθε αναδρομική κλήση ένα σφάλμα ή υποστόχο προς επίλυση, η διαδικασία POP έχει ξεχωριστό έλεγχο για την επίλυση των απειλών και των υποστόχων. Σε κάθε αναδρομική κλήση πρώτα μεταβάλλει το πλάνο με βάση την επίτευξη ενός υποστόχου και στη συνέχεια επιλύει τυχόν ασυνέπειες και απειλές που έχουν προκύψει με βάση τη λύση για τον υποστόχο που έχει επιλέξει. Στην ουσία, η διαδικασία ανάγεται σε δύο απλά βήματα. Το ένα είναι η επιλογή της κατάλληλης ενέργειας για την επίτευξη του υποστόχου και το άλλο είναι η εισαγωγή ενός περιορισμού σειράς ή περιορισμού δέσμευσης μεταβλητών για την εξουδετέρωση μιας απειλής.

Η οπισθοδρόμηση σε περίπτωση αποτυχίας πραγματοποιείται χρονολογικά μεταβαίνοντας σε όλες τις αλλαγές που έχουν γίνει για την επίλυση των απειλών και σε περίπτωση που αυτές εξαντληθούν, οπισθοδρομεί σε άλλη ενέργεια για τη λύση του δεδομένου υποστόχου.

Στην εικόνα 4 παρουσιάζεται ο αλγόριθμος της διαδικασίας POP. Παρατηρούμε ότι ο αλγόριθμος δέχεται δύο ορίσματα. Το τρέχον πλάνο και μία μεταβλητή που ονομάζεται agenda. Αυτή η μεταβλητή περιέχει ένα σύνολο ταξινομημένων ζευγαριών (a,p), όπου

το a αντιπροσωπεύει μία ενέργεια ενώ το p είναι μια προϋπόθεση της a που θεωρείται υποστόχος. Στον αλγόριθμο υπάρχει ακόμα η συνάρτηση Providers() η οποία επιστρέφει το σύνολο των ενεργειών που περιέχουν μία επίδραση η οποία μπορεί να ενοποιηθεί με την προϋπόθεση p. Το σύνολο αυτών των ενεργειών θεωρείται σχετικό για τον τρέχων υποστόχο.

PoP(π, *agenda*) ;; where $\pi = (A, \prec, B, L)$
  if *agenda* $= \emptyset$ then return(π)
  select any pair (a_j, p) in and remove it from *agenda*
  relevant $\leftarrow$ Providers(p, π)
  if *relevant* $= \emptyset$ then return(failure)
  nondeterministically choose an action $a_i \in$ *relevant*
  $L \leftarrow L \cup \{\langle a_i \xrightarrow{p} a_j \rangle\}$
  update B with the binding constraints of this causal link
  if a_i is a new action in A then do:
    update A with a_i
    update $\prec$ with $(a_i \prec a_j), (a_0 \prec a_i \prec a_\infty)$
    update *agenda* with all preconditions of a_i
  for each threat on $\langle a_i \xrightarrow{p} a_j \rangle$ or due to a_i do:
    resolvers $\leftarrow$ set of resolvers for this threat
    if *resolvers* $= \emptyset$ then return(failure)
    nondeterministically choose a resolver in *resolvers*
    add that resolver to $\prec$ or to B
  return(PoP(π, *agenda*))
end

Εικόνα 4. Αλγόριθμος διαδικασίας POP

2.3.5 Διαφορές ανάμεσα στις κατηγορίες του κλασσικού σχεδιασμού

Η κύρια διαφορά ανάμεσα σε αυτές τις δύο κατηγορίες είναι ο τρόπος που εκτελείται η αναζήτηση. Στη περίπτωση του χώρου καταστάσεων, η επιλογή μιας ενέργειας επιφέρει αλλαγές στη τωρινή κατάσταση, η οποία ουσιαστικά αποτελεί μια ισχυρή δέσμευση στην ακολουθία των ενεργειών που πρέπει να εκτελεστούν. Στη περίπτωση που αυτή η ακολουθία δεν οδηγήσει σε κατάσταση στόχου, είναι απαραίτητη η οπισθοδρόμηση ώστε να εκτελεστεί μια εναλλακτική επιλογή, η οποία με τη σειρά της αποτελεί νέα ισχυρή δέσμευση. Αντίστοιχα, στο χώρο πλάνων έχουμε επίσης επιλογές/δεσμεύσεις κατά τη διάρκεια της αναζήτησης, αλλά αυτές δεν αφορούν την ακολουθία των ενεργειών, αλλά τους αιτιολογικούς συνδέσμους μεταξύ των ενεργειών. Έτσι, δεν προκαθορίζεται μια αλληλουχία εκτέλεσης των ενεργειών, αλλά μόνο περιορισμοί στις σχέσεις αυτών.

Από την άλλη πλευρά ο σχεδιασμός με αναζήτηση στο χώρο πλάνων έχει κάποια πλεονεκτήματα τα οποία δεν μας τα παρέχει ο σχεδιασμός με αναζήτηση στο χώρο καταστάσεων. Η δημιουργία μερικώς ορισμένων πλάνων είναι πιο σαφή και ευέλικτα κατά την εκτέλεση από τα πλάνα που έχουν δημιουργηθεί κατά το σχεδιασμό με αναζήτηση στο χώρο καταστάσεων. Μπορούμε πολύ εύκολα να επεκτείνουμε τους υπάρχοντες αλγορίθμους και να δημιουργήσουμε πληθώρα πρόσθετων προγραμμάτων τα οποία να χειρίζονται καθώς και να εισάγουμε νέα είδη περιορισμών σαν γενίκευση της διαδικασίας PSP. Τέλος, εισάγουν πολύ φυσικά και απλά την ιδέα πολλαπλών καταστρωτών σχεδίου οι οποίοι μπορούν να δρουν πάνω στον ίδιο χώρο αναζήτησης αφού μπορούμε να χρησιμοποιήσουμε διαφορετικές τεχνικές για την ένωση επιμέρους πλάνων τα οποία μπορεί να διαχειριστεί εύκολα η δομή της διαδικασίας POP (partial planning)

Καταλήγοντας, στη σημερινή εποχή οι καταστρωτές σχεδίου με αναζήτηση στο χώρο πλάνων δεν είναι ιδιαίτερα αποδοτικοί από την άποψη της υπολογιστικής πολυπλοκότητας που χρειάζονται για την εύρεση του πλάνου. Αυτό συμβαίνει γιατί, οι καταστρωτές σχεδίου με αναζήτηση στο χώρο καταστάσεων μπόρεσαν να εκμεταλλευτούν τις σαφώς ορισμένες καταστάσεις και δημιούργησαν ισχυρές ευρετικές συναρτήσεις που τους επιτρέπει να μειώσουν τον όγκο αναζήτησης οποιουδήποτε μεγάλου προβλήματος.

2.3.6 Νέο-κλασσικός σχεδιασμός

Αξίζει να αναφερθεί πως τα τελευταία χρόνια στο σχεδιασμό με αναζήτηση στο χώρο καταστάσεων έχουν αρχίσει να χρησιμοποιούνται πιο εξεζητημένες τεχνικές οι οποίες έχουν την ικανότητα να μειώσουν αρκετά το χώρο αναζήτησης. Τέτοιες τεχνικές είτε χρησιμοποιούν, για παράδειγμα ένα γράφο για να εισάγουν περιορισμούς στο χώρο αναζήτησης αφαιρώντας τις καταστάσεις που δεν τους ικανοποιούν και κατ' επέκταση μειώνοντας το δραστικά (GraphPlan) [3], είτε αναγάγουν το δεδομένο πρόβλημα σχεδιασμού σε ένα αντίστοιχο πρόβλημα ικανοποιησιμότητας για την πιο αποτελεσματική επίλυσή του (SatPlan) [4].

Στην επόμενη ενότητα θα αναπτύξουμε το σχεδιασμό με τη χρήση ιεραρχικών δικτύων διαδικασιών.

2.3.7 Ιεραρχικά Δίκτυα Διαδικασιών (HTN)

Ο σχεδιασμός με ιεραρχικά δίκτυα διαδικασιών [5] είναι παρόμοιος με τον κλασσικό σχεδιασμό στο γεγονός ότι οι καταστάσεις παρουσιάζονται ως ένα σύνολο από λεκτικά και κάθε ενέργεια αντιστοιχεί σε μία ντετερμινιστική μετάβαση από μία κατάσταση σε μία άλλη. Η βασική διαφορά έγκειται στο στόχο της κατάστρωσης σχεδίου και στον τρόπο με τον οποίο πραγματοποιούν το σχεδιασμό.

Ένας καταστρωτής σχεδίου που χρησιμοποιεί HTN έχει ως στόχο να εκτελέσει μία σειρά διαδικασιών και όχι να ικανοποιήσει μία σειρά από στόχους όπως ο κλασσικός σχεδιασμός. Η είσοδος στους αλγορίθμους HTN είναι σύνολο από τελεστές παρόμοιους με αυτούς του κλασσικού σχεδιασμού (STRIPS) και ένα σύνολο από μεθόδους που παρέχουν οδηγίες για τη διάσπαση πολύπλοκών διαδικασιών σε μικρότερες. Η κατάστρωση σχεδίου ξεκινάει από την αρχική διαδικασία και με τη χρήση των μεθόδων τη διασπά σε μικρότερες. Σταδιακά, όλες οι μη-θεμελιώδεις διαδικασίες διασπώνται μέχρις ότου να υπάρξουν μόνο θεμελιώδεις διαδικασίες οι οποίες μπορούν να εκτελεστούν με τη χρήση των τελεστών σχεδιασμού (ενεργειών).

Αναλύοντας τα ιεραρχικά δίκτυα διαδικασιών παρατηρούμε ότι απαρτίζεται από τα ακόλουθα στοιχεία:

1. Το δίκτυο διαδικασιών
2. Τις μεθόδους HTN
3. Το πεδίο σχεδιασμού HTN (HTN domain) και τα προβλήματα HTN (HTN problems)

4. Τις διαδικασίες σχεδιασμού.

Το δίκτυο διαδικασιών είναι ένα ζευγάρι (U,C) που το U αντιπροσωπεύει ένα σύνολο από κόμβους διαδικασιών (διαδικασίες) ενώ το C αντιπροσωπεύει ένα σύνολο από περιορισμούς πάνω στις διαδικασίες. Κάθε περιορισμός στο σύνολο C περιλαμβάνει μία προϋπόθεση που πρέπει να πληροί το πλάνο λύσης του προβλήματος.

Υπάρχουν πολλών ειδών περιορισμοί που διαχειρίζονται τα ιεραρχικά δίκτυα και μία γενική κατηγοριοποίηση είναι η ακόλουθη:

- Οι περιορισμοί προτεραιοτήτων ανάμεσα σε δύο διαδικασίες u,v σημαίνει ότι σε κάθε πλάνο σχεδιασμού η τελευταία ενέργεια της διαδικασίας u θα γίνει αναγκαστικά πριν από την πρώτη ενέργεια της διαδικασίας v.
- Οι πριν την εκτέλεση περιορισμοί (before constraints) είναι της μορφής before(U',l) όπου το U' είναι ένα υποσύνολο των διαδικασιών του U και το l είναι ένα λεκτικό. Αυτό δηλώνει πως το λεκτικό l πρέπει να είναι αληθές στην κατάσταση πριν πραγματοποιηθεί η πρώτη ενέργεια του συνόλου διαδικασιών U'.
- Οι μετά την εκτέλεση περιορισμοί (after constraints) είναι της μορφής after(U',l) και είναι αντίστοιχη με τους πριν την εκτέλεση περιορισμούς μόνο που στη συγκεκριμένη περίπτωση το λεκτικό l πρέπει να είναι αληθές στην κατάσταση που θα προκύψει μετά την εκτέλεση των ενεργειών του συνόλου διαδικασιών U'.
- Οι ενδιάμεσοι περιορισμοί (between constraints) οι οποίοι είναι της μορφής between(U',U'',l) και δηλώνουν ότι το λεκτικό l πρέπει να είναι αληθές στην κατάσταση που προκύπτει μετά την εκτέλεση της τελευταίας ενέργειας του συνόλου διαδικασιών U' και πριν την εκτέλεση της πρώτης ενέργειας του συνόλου διαδικασιών U''.

Οι HTN μέθοδοι είναι μία τετράδα της μορφής

$$m = (name(m), task(m), subtasks(m), constr(m))$$

- Το όνομα μεθόδου (name) είναι μία έκφραση της μορφής $n(x_1,x_2,\ldots..,x_k)$ όπου το n είναι ένα σύμβολο μοναδικής περιγραφής της μεθόδου και τα $x_1,x_2,\ldots..,x_k$ είναι το σύνολο των μεταβλητών που εμφανίζονται και χρησιμοποιούνται μέσα στη μέθοδο m.
- Η διαδικασία (task) είναι μία μη θεμελιώδης διαδικασία.
- Το ζευγάρι (*subtasks(m),constr(m))* αποτελεί ένα δίκτυο διεργασιών.

Ας υποθέσουμε ότι έχουμε έναν κόμβο διαδικασιών u και μία μέθοδο m και ότι η μέθοδος m διασπά τον κόμβο διαδικασιών u σε subtasks(m) τότε παράγεται ένα καινούριο δίκτυο διαδικασιών του οποίο το σύνολο των καινούριων διαδικασιών αντιστοιχεί στην ένωση του παλιού συνόλου U με το subtasks(m) αφαιρώντας το σύνολο u, ενώ οι περιορισμοί αντιστοιχούν στην ένωση του συνόλου περιορισμών C' με το *constr(m).*

Το σύνολο περιορισμών C' είναι αντίστοιχο με το σύνολο περιορισμών C αφού έχουν γίνει οι απαραίτητες μεταβολές με βάση τη διάσπαση. Η πρώτη μεταβολή είναι η αντικατάσταση κάθε περιορισμού προτεραιοτήτων που περιείχε το σύνολο u με τους αντίστοιχους περιορισμούς προτεραιοτήτων που περιέχουν οι κόμβοι του συνόλου subtasks(m). Η δεύτερη μεταβολή είναι η αντικατάσταση περιορισμών τύπου πριν από την εκτέλεση (before), ανάμεσα (between) και μετά (after), οι οποίοι περιέχουν αναφορά στο σύνολο u με το αντίστοιχο που προέκυψε στο μετά τη δημιουργία του νέου δικτύου διαδικασιών.

Το πεδίο σχεδιασμού HTN (HTN domain) είναι ένα ζευγάρι (O, M) όπου O είναι ένα σύνολο ενεργειών και M είναι ένα σύνολο μεθόδων, ενώ τα προβλήματα HTN αντιστοιχούν σε μία τετράδα (s_0,w,O,M) όπου s_0 είναι η αρχική κατάσταση, w το αρχικό δίκτυο διαδικασιών ενώ τα O και M είναι αντίστοιχα με αυτά του πεδίου σχεδιασμού.

Τη λύση για ένα πρόβλημα σχεδιασμού μπορούμε να τη χωρίσουμε σε δύο κατηγορίες:

- Αν το δίκτυο διαδικασιών περιέχει μόνο θεμελιώδεις διαδικασίες, τότε το πλάνο που έχουμε δημιουργήσει μέχρι τη δεδομένη στιγμή είναι λύση του προβλήματος σχεδιασμού και ταυτόχρονα ακολουθεί τους παρακάτω κανόνες:
    1) Οι ενέργειες του πλάνου καθορίζονται από τους κόμβους u_1, u_2, u_3,….,u_k που υπάρχουν στο w.
    2) Το πλάνο είναι εκτελέσιμο από την αρχική κατάσταση s_0
    3) Ικανοποιούνται όλοι οι περιορισμοί που υπάρχουν στο δεύτερο σκέλος του δικτύου διαδικασιών.
- Αν το δίκτυο διαδικασιών περιέχει και μη-θεμελιώδεις διαδικασίες, τότε το τρέχον πλάνο μπορεί να θεωρηθεί λύση σε δεδομένο πρόβλημα σχεδιασμού εφόσον υπάρχει μία αλληλουχία από διαδικασίες η οποία να διασπά τις μη-θεμελιώδεις διαδικασίες μέχρι να δημιουργηθεί ένα δίκτυο θεμελιωδών διαδικασιών. Σε αυτή την περίπτωση, το τρέχον πλάνο που έχουμε πρέπει να αποτελεί λύση για το νέο δίκτυο (θεμελιωδών) διαδικασιών.

Οι διαδικασίες σχεδιασμού πρέπει να είναι σε θέση να αρχικοποιούν ενέργειες και να διασπούν διαδικασίες. Επειδή υπάρχει πληθώρα τρόπων ως προς το σχεδιασμό και την υλοποίηση και των δύο διαδικασιών μπορούμε να χρησιμοποιήσουμε έναν γενικευμένο αλγόριθμο που περιλαμβάνει τη βασική ιδέα των HTN.

```
Abstract-HTN(s, U, C, O, M)
    if (U, C) can be shown to have no solution
        then return failure
    else if U is primitive then
        if (U, C) has a solution then
            nondeterministically let π be any such solution
            return π
        else return failure
    else
        choose a nonprimitive task node u ∈ U
        active ← {m ∈ M | task(m) is unifiable with t_u}
        if active ≠ ∅ then
            nondeterministically choose any m ∈ active
            σ ← an mgu for m and t_u that renames all variables of m
            (U', C') ← δ(σ(U, C), σ(u), σ(m))
            (U', C') ← apply-critic(U', C') ;; this line is optional
            return Abstract-HTN(s, U', C', O, M)
        else return failure
```

Εικόνα 5. Ο γενικευμένος αλγόριθμος για τα HTN.

Στην εικόνα 5 παρατηρούμε ότι η επιλογή διαδικασίας προς διάσπαση είναι ντετερμινιστική, αφού όλες οι διαδικασίες θα πρέπει να διασπαστούν και να καταλήξουν σε θεμελιώδεις. Επειδή το μικρότερο δένδρο διάσπασης διαδικασιών μπορεί να είναι κατά πολλές φορές μικρότερο σε μέγεθος από το μεγαλύτερο δένδρο διάσπασης διαδικασιών, ένας καλός αλγόριθμος για σχεδιασμό με τη χρήση HTN πρέπει να προσπαθεί να επιλέξει μία διάσπαση που να έχει όσο το δυνατόν μικρότερο δένδρο διαδικασιών.

Ο σχεδιασμός με τη χρήση ιεραρχικών δικτύων διαδικασιών χρησιμοποιείται για πρακτικές εφαρμογές περισσότερο από οποιοδήποτε άλλο είδος σχεδιασμού αρχικά λόγω της ευελιξίας του και δεύτερον λόγο της καλής κλιμάκωσης του χώρου αναζήτησης. Η ευελιξία επιτρέπει στους προγραμματιστές να προσαρμόσουν πληθώρα προβλημάτων σχεδιασμού και να εξάγουν λύσεις εύκολα κατανοητές, οι οποίες

αντιστοιχούν στις λύσεις που θα έδιναν αναλυτές για ένα αντίστοιχο πρόβλημα σχεδιασμού. Η κλιμάκωση του χώρου αναζήτησης βοηθάει γιατί εμπλουτίζει το χώρο αναζήτησης της κατάστρωσης σχεδίου με νέα γνώση κάνοντας την αναζήτηση πιο αποτελεσματική.

Πρέπει εδώ να τονιστεί ότι ακόμα και σε αυτή τη μέθοδο σχεδιασμού μπορεί να χρησιμοποιηθεί μία γλώσσα σαν τη STRIPS για την περιγραφή των καταστάσεων, των ενεργειών και τον περιορισμών.

Τα ιεραρχικά δίκτυα, αν και είναι η πιο συχνά χρησιμοποιημένη μέθοδος σχεδιασμού στα πλαίσια της παρούσας πτυχιακής, μελετήθηκαν μόνο θεωρητικά και παρουσιάζονται στον αναγνώστη για να αποκτήσει μία ολοκληρωμένη αντίληψη για τις πιο γνωστές μεθόδους σχεδιασμού. Στην παρούσα πτυχιακή ασχοληθήκαμε μόνο με τη μελέτη και την υλοποίηση μεθόδων κλασσικού σχεδιασμού και τη μελέτη της STRIPS.

2.4 Ακαδημαϊκοί σχεδιαστές

Στις προηγούμενες ενότητες μελετήσαμε αρκετές μεθόδους σχεδιασμού, χωρίς όμως να αναφερθούμε στα εργαλεία που τις αξιοποιούν για να λύσουν πρακτικά προβλήματα. Στις επόμενες υποενότητες θα παρουσιασθούν μερικά από τα πιο γνωστά συστήματα που έχουν αναπτυχθεί από την ακαδημαϊκή κοινότητα και που λειτουργούν ως χαρακτηριστικές υλοποιήσεις.

Όπως αναφέρθηκε και νωρίτερα, υπάρχει πληθώρα υποθέσεων που μπορούν να χρησιμοποιηθούν για τη περιγραφή των προβλημάτων σχεδιασμού και για την εύρεση λύσης. Αυτό οδηγεί σε πολλά διαφορετικά συστήματα σχεδιαστών, τα οποία μπορεί:

- να εστιάζουν σε συγκεκριμένους τύπους προβλημάτων, ή να είναι γενικού τύπου
- να κάνουν υποθέσεις για τον κόσμο, ή να θεωρούν πως ο κόσμος τους είναι στατικός
- να λειτουργούν με ένα στόχο, ή με σύνολο στόχων
- να έχουν ολοκληρωμένη γνώση για το περιβάλλον, να εξελίσσουν σταδιακά αυτή τη γνώση, ή ακόμη και να λειτουργούν με αβεβαιότητα
    - χρησιμοποιώντας πιθανώς και μεθόδους μηχανικής μάθησης (machine learning)
- και, φυσικά, να χρησιμοποιούν κάποιον από τους πολλούς διαθέσιμους αλγορίθμους κατάστρωσης σχεδίου, ή και συνδυασμό αυτών.

Όπως είναι φυσικό, τέτοια συστήματα έχουν διαφορετικές δυνατότητες και επιδόσεις ανά τύπο προβλήματος. Για να μπορεί να αξιολογηθεί η επίδοσή τους, έχουν αναπτυχθεί γλώσσες περιγραφής προβλημάτων σχεδιασμού, όπως η PDDL [6], οι οποίες αποτελούν κοινό πρότυπο και υποστηρίζονται από τα περισσότερα συστήματα σχεδιασμού. Επιπλέον, υπάρχουν διαγωνισμοί όπου περιγράφονται συγκεκριμένοι κόσμοι και προβλήματα σχεδιασμού τα οποία καλούνται να επιλύσουν οι σχεδιαστές. Αυτοί αποτελούν και το κοινά αποδεκτό κριτήριο αξιολόγησης των συστημάτων. Ειδικά για τους planners που δέχονται ως είσοδο περιγραφές προβλημάτων στη γλώσσα PDDL, υπάρχει ο διεθνής διαγωνισμός International Planning Competition[4]. Στις επόμενες ενότητες θα περιγράψουμε τρεις από γνωστότερους σχεδιαστές.

2.4.1 Κλασσικοί Σχεδιαστές (Classical planners)

2.4.1.1 Fast Forward

Ο σχεδιαστής FF [7] και αποτελεί ένα σύστημα σχεδιασμού, που μπορεί να χειριστεί προβλήματα που περιγράφονται σε STRIPS ή σε PDDL και που λειτουργεί ανεξάρτητα από τη γνώση που περιέχει η περιγραφή του προβλήματος.

Ο FF χρησιμοποιεί προς τα εμπρός αναζήτηση με τη χρήση ευρετικής συνάρτησης. Η ευρετική συνάρτηση βασίζεται στην ιδέα ενός παλαιότερού σχεδιαστή, του HSP. Για να εκτιμήσει το κόστος που χρειάζεται από μία κατάσταση για την κατάσταση στόχου επιλύει ένα χαλαρό πρόβλημα για τη δεδομένη κατάσταση και με τη χρήση ενός αλγορίθμου GraphPlan. Το σύνολο των ενεργειών που απαιτούνται για την επίλυση του χαλαρού προβλήματος το χρησιμοποιεί ως κόστος της δεδομένης κατάστασης. Για την επίλυση του προβλήματος ο αλγόριθμος χρησιμοποιεί τη μέθοδο της εξαναγκασμένης αναζήτησης λόφου (enforced hill climbing[5]).

Ο αλγόριθμος αυτός τιμήθηκε με τον τίτλο «Group A distinguished performance Planning System» στο 2^{o} Διεθνή Διαγωνισμό για Σχεδιαστές και με το βραβείο Schindler για την καλύτερη απόδοση συστήματος σχεδιασμού στο Miconic 10 Elevator domain, ADL track.

2.4.1.2 Blackbox

Ο Blackbox [8] μετατρέπει προβλήματα που έχουν περιγραφεί σε STRIPS ή PDDL σε προβλήματα ικανοποιησιμότητας. Τα προβλήματα αυτά τα επεξεργάζεται αρχικά με τη

[4] http://ipc.icaps-conference.org/

[5] http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/node5.html

χρήση GraphPlan για τη μείωση του χώρου αναζήτησης και στη συνέχεια τα επιλύει με τη χρήση ενός SAT solver.

Δεδομένου ότι ο Blackbox δεν χρησιμοποιεί συγκεκριμένο SAT solver αλλά μπορεί να αξιοποιήσει οποιαδήποτε τέτοια μέθοδο ακόμα και κατά την εκτέλεση του ίδιου προβλήματος τον καθιστά πολύ ευέλικτο και αποτελεσματικό για πληθώρα προβλημάτων σχεδιασμού.

2.4.2 Σχεδιαστές ιεραρχικών δικτύων (HTN planners)

2.4.2.1 SHOP2

Ο SHOP2 [9] αποτελεί εργαλείο σχεδιασμού που αξιοποιεί ιεραρχικά δίκτυα διαδικασιών (HTN) για την επίλυση προβλημάτων σχεδιασμού και είναι ο πιο χαρακτηριστικός σχεδιαστής της κατηγορίας του. Επιλύει προβλήματα τα οποία περιγράφονται σε STRIPS αλλά και PDDL.

Ο SHOP2 χρησιμοποιεί μία μέθοδο ταξινομημένης διάσπασης διαδικασιών κατά την οποία όλες οι ενέργειες που χρειάζονται για την εκπλήρωση ενός πλάνου παράγονται με τη σειρά που θα εκτελεστούν και έχει ως αποτέλεσμα να είναι γνωστές οι καταστάσεις που παράγονται σε κάθε βήμα της διαδικασίας σχεδιασμού. Αυτή η διαδικασία μειώνει δραστικά την πολυπλοκότητα της επίλυσης του προβλήματος αφού αφαιρεί μεγάλη αβεβαιότητα για το περιβάλλον και τον κόσμο που δρα ο αλγόριθμος, καθιστώντας αρκετά εύκολο να ενσωματωθούν επιμέρους εκφραστικές δυνατότητες στο σύστημα σχεδιασμού.

Ο αλγόριθμος αυτός κέρδισε ένα από τα τέσσερα κορυφαία βραβεία καθώς και ένα από τα δύο βραβεία για καλύτερη επίδοση στο Διεθνή Διαγωνισμό Σχεδιασμού το 2002.

2.5 Κατάστρωση σχεδίου σε ηλεκτρονικά παιχνίδια (Planners in videogames)

2.5.1 Σύγκριση με τους ακαδημαϊκούς σχεδιαστές

Οι σχεδιαστές που υλοποιούνται για τη χρήση σε ηλεκτρονικά παιχνίδια διαφέρουν από τους ακαδημαϊκούς σχεδιαστές σε πολλά σημεία. Τα βασικότερα σημεία είναι η λειτουργικότητα και οι επιδόσεις. Ενώ οι ακαδημαϊκοί σχεδιαστές δημιουργούνται για την επίλυση πληθώρας προβλημάτων, οι σχεδιαστές για χρήση σε ηλεκτρονικά παιχνίδια δημιουργούνται για πολύ συγκεκριμένο σκοπό και συνήθως μόνο για τις ανάγκες του παιχνιδιού για το οποίο σχεδιάζονται, αντίστοιχα με εμπορικούς σχεδιαστές που επιλύουν αποδοτικά πολύ συγκεκριμένους τύπους προβλημάτων.

Οι επιδόσεις θεωρούνται ακόμα μία από τις βασικότερες διαφορές τους. Ένας σχεδιαστής που χρησιμοποιείται σε ηλεκτρονικά παιχνίδια πρέπει σε κάθε περίπτωση να παράγει μέσα σε λίγα στιγμιότυπα ενός παιχνιδιού ένα πλάνο που να μπορεί να χρησιμοποιηθεί από τους πράκτορες του παιχνιδιού. Από την άλλη πλευρά υπάρχει μία σχετική ελαστικότητα για την απόδοση ενός ακαδημαϊκού σχεδιαστή, μιας και καθορίζεται από την απόφαση του εκάστοτε αναπτυκτή λογισμικού.

Στη συνέχεια θα αναπτύξουμε δύο βασικές τεχνικές σχεδιασμού που χρησιμοποιούνται σήμερα εκτενώς σε ηλεκτρονικά παιχνίδια. Η μία υλοποιεί μεθόδους του κλασσικού σχεδιασμού και ονομάζεται GOAP (Goal-Oriented Action Planning) και η άλλη χρησιμοποιεί ιεραρχικά δένδρα διεργασιών και ονομάζεται Behavior Trees.

2.5.2 GOAP

Αν και η γλώσσα STRIPS αποτελεί πολύ χρήσιμη απλοποίηση της διαδικασίας σχεδιασμού σε ακαδημαϊκούς σχεδιαστές, οι ιδιαίτερες ανάγκες των βιντεοπαιχνιδιών απαιτούν σχεδιαστικές επιλογές που εισάγουν περιπλοκότητα στην αρχιτεκτονική και σχεδίαση των συστημάτων κατάστρωσης σχεδίου.

Ιδανικό παράδειγμα είναι το σύστημα GOAP[6], το οποίο υλοποιεί ένα σύστημα βασισμένο στο STRIPS με αρκετά πιο απλοποιημένη σχεδιαστική διαδικασία και με χαρακτηριστικά που συμβαδίζουν με την ιδιαίτερη αρχιτεκτονική των πρακτόρων. Στο σύστημα GOAP προσθέτει κόστος ανά ενέργεια, περιλαμβάνει μια ενιαία συλλογή επιδράσεων, ενώ οι προϋποθέσεις και επιδράσεις είναι σχετικές με το άμεσο περιβάλλον εφαρμογής.

Ένα επιπλέον χαρακτηριστικό του συστήματος GOAP είναι πως υποστηρίζει ομαδοποίηση και διαμοιρασμό ενεργειών σε πολλούς πράκτορες, απλοποιώντας σημαντικά την οργάνωση της διαδικασίας σχεδιασμού. Σε συνδυασμό με αρχιτεκτονική μαυροπίνακα (blackboard) για τη φύλαξη της γνώσης και σύστημα τακτικής ομαδοποίησης πρακτόρων, το GOAP εξοικονομεί πόρους κατά τη διάρκεια της κατάστρωσης σχεδίου.

Αυτά τα χαρακτηριστικά, σε συνδυασμό με βελτιστοποιήσεις στη κατάστρωση σχεδίου όπως ο περιορισμός του πλήθους των ενεργειών, του δυνατού βάθους πλάνου και τη χρήση τυποποιημένων συμπεριφορών σε περίπτωση αποτυχίας ταχείας εύρεσης

[6] http://web.media.mit.edu/~jorkin/goap.html

λύσης, οδηγούν σε ένα σύστημα κατάλληλο για το ταχύτατα εναλλασσόμενο και απαιτητικό περιβάλλον ενός βιντεοπαιχνιδιού.

Η συγκεκριμένη αρχιτεκτονική αναπτύχθηκε και χρησιμοποιήθηκε στα πλαίσια του βιντεοπαιχνιδιού F.E.A.R από τον Jeff Orkin, και αποτελεί χαρακτηριστική πλατφόρμα μελέτης της κατάστρωσης σχεδίου σε περιβάλλοντα βιντεοπαιχνιδιών, καθώς και πηγή έμπνευσης για πληθώρα σχετικών εργασιών.

2.5.3 Behavior Trees

Τα δένδρα συμπεριφοράς (Behavior trees)[7] έχουν γίνει τα τελευταία χρόνια πολύ δημοφιλή για τη δημιουργία ΤΝ χαρακτήρων σε ηλεκτρονικά παιχνίδια. Χαρακτηριστικό παράδειγμα ενός διάσημου παιχνιδιού που χρησιμοποίησε εκτενώς δένδρα συμπεριφοράς είναι το Halo 2 και στη συνέχεια υιοθέτησαν αυτή την τεχνική πολλά μεταγενέστερα παιχνίδια.

Η βασική ιδέα πίσω από αυτή τη μέθοδο σχεδιασμού είναι παρόμοια με αυτή των ιεραρχικών μηχανών καταστάσεων (hierarchical state machines) με τη διαφορά ότι αντί για καταστάσεις έχουν διαδικασίες (tasks). Μία διαδικασία μπορεί να είναι κάτι πολύ απλό όπως η αναζήτηση μίας τιμής σε μία κατάσταση ή κάτι αρκετά πολύπλοκο όπως η παρουσίαση και απεικόνιση μιας κίνησης (animation).

Σύνολα τέτοιων διαδικασιών οργανώνονται σε υποδένδρα και παράγουν πιο πολύπλοκες ενέργειες. Αυτές με τη σειρά τους παράγουν υψηλού επιπέδου συμπεριφορές που αξιοποιούνται από τους πράκτορες του παιχνιδιού. Αυτή η ικανότητα να συνδυάζονται διαδικασίες είναι και το πλεονέκτημα που έχουν τα δένδρα συμπεριφοράς αφού όλες οι διαδικασίες έχουν κοινό τρόπο οργάνωσης και μπορούν να οργανωθούν σε οποιαδήποτε διεργασία.

Όπως αναφέρθηκε οι διαδικασίες μπορεί να είναι πολλών διαφορετικών ειδών, οπότε είναι πιθανό να περιέχουν πολύπλοκο κώδικα. Για να γίνουν λοιπόν πιο ευέλικτες, είναι δόκιμο να διασπαστούν σε μικρότερες διαδικασίες τις οποίες να μπορέσει η κατάστρωση σχεδίου να αξιοποιήσει στο συνδυασμό μεγαλύτερων και πολύπλοκων ενεργειών. Για τη διάσπαση αυτή καθώς και για το συνδυασμό σε πολύπλοκες ενέργειες είναι ιδιαίτερα αποτελεσματικός ο σχεδιασμός με τη χρήση των ιεραρχικών δικτύων διαδικασιών (HTN) όπως συζητήθηκε σε προηγούμενη ενότητα.

---

[7] http://aigamedev.com/open/article/bt-overview/

3. Planning σε Unity3D (iThink Planner)

3.1 Γενική περιγραφή Unity3D

Το Unity3D είναι μια πλατφόρμα εργαλείων και μηχανή παιχνιδιών που προσφέρει μια πληθώρα ευκολιών που την καθιστούν ιδανική για χρήση ως πλατφόρμα ταχείας ανάπτυξης και ελέγχου πρωτοτύπων.

Σκοπός της πτυχιακής μας ήταν η μελέτη της κατάστρωσης σχεδίων σε διαδραστικό περιβάλλον βιντεοπαιχνιδιών και η ευκολότερη οπτικοποίηση σχεδίων, οπότε η χρήση της Unity3D μας αποδέσμευε από τις λεπτομέρειες και περιπλοκότητες χρήσης ή και ανάπτυξης κάποιας πλατφόρμας, καθιστώντας και δυνατή την υλοποίηση μιας βιβλιοθήκης λογισμικού που θα είναι άμεσα διαχειρίσιμη από άλλους αναπτυκτές λογισμικού.

Επιπλέον, λόγω της στήριξής της στη πλατφόρμα Mono (ελεύθερη υλοποίηση του .Net framework), η Unity3D είναι δυνητικά μεταφέρσιμη σε πολλά διαφορετικά συστήματα και μπορεί να υποστηρίξει διαφορετικές γλώσσες προγραμματισμού. Στη παρούσα φάση, υποστηρίζονται οι γλώσσες JavaScript, Boo (διάλεκτος της Python) και C# 2.0.

Λόγω λοιπόν αυτής της λειτουργίας των γλωσσών της πλατφόρμας, ο χρήστης μπορεί να φτιάξει αρχεία κώδικα (που η Unity3D αποκαλεί scripts) τα οποία αποτελούν είτε δέσμες ενεργειών μεταφραζόμενων (scripting) γλωσσών προγραμματισμού (όπως στη περίπτωση των JavaScript και Boo) ή πηγαίο κώδικα αντικειμενοστραφών γλωσσών, όπως η C#, με ισοδύναμη όμως εκφραστικότητα.

Η χρήση της πλατφόρμας είναι δωρεάν για το βασικό σύνολο χαρακτηριστικών της και σε συνδυασμό με τη μεταφερσιμότητά της την καθιστά άμεσα διαθέσιμη σε πληθώρα αναπτυκτών λογισμικού και χρηστών.

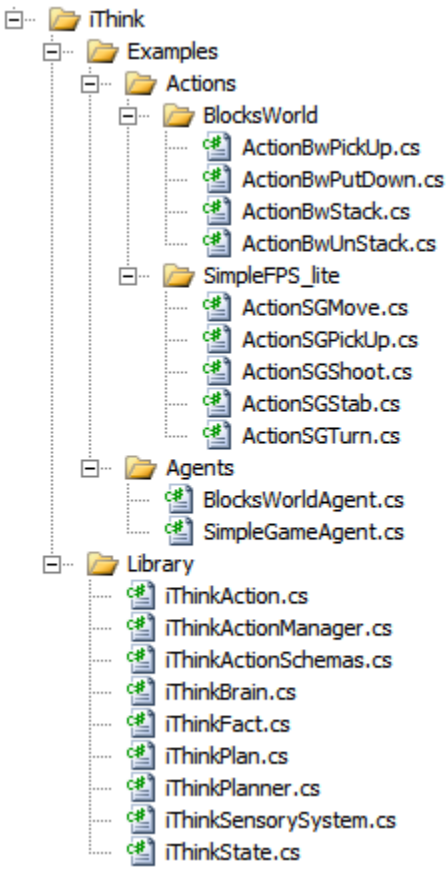
3.2 Γενική περιγραφή iThink

Η iThink είναι μια ανοικτού λογισμικού βιβλιοθήκη που υλοποιεί έναν κλασικό σχεδιαστή στο περιβάλλον Unity3D. Οι γνωστότερες εργασίες που ασχολούνται με κατάστρωση σχεδίου σε βιντεοπαιχνίδια στηρίζονται στον GOAP σχεδιαστή του βιντεοπαιχνιδιού F.E.A.R., ο οποίος εκμεταλλεύεται τη μορφή του συγκεκριμένου περιβάλλοντος για να λειτουργεί σε πραγματικό χρόνο με μεγάλο πλήθος πρακτόρων, θυσιάζοντας αναγνωσιμότητα και ευρύτερη λειτουργικότητα ώστε να αξιοποιήσει πληθώρα βελτιστοποιήσεων.

Αντιθέτως, ο δικός μας σχεδιαστής αποσκοπεί στην ευκολία χρήσης, στην αναγνωσιμότητα και στη χρήση σε οποιοδήποτε περιβάλλον/πρόβλημα, ώστε να απαιτεί την ελάχιστη δυνατή συνεισφορά από το χρήστη, λειτουργώντας όμως και σαν βάση για τη δημιουργία ισχυρότερων σχεδιαστών προσαρμοσμένων στις ανάγκες του προγραμματιστή εάν αυτός το επιθυμεί. Στη παρούσα πτυχιακή ο στόχος αυτός επιτεύχθηκε και μάλιστα συζητούνται αρκετοί τύποι βελτιστοποιήσεων που μπορούν να αξιοποιηθούν σε μελλοντικές εργασίες και τις οποίες επιθυμούμε να υλοποιήσουμε στο άμεσο μέλλον.

3.3 Δομή βιβλιοθήκης iThink

Ο φάκελος iThink παρέχει τον πυρήνα της υλοποίησης, ο οποίος αποτελείται από τα αρχεία της βασικής βιβλιοθήκης στο φάκελο *«Library»* και το φάκελο των παραδειγμάτων *«Examples»* που περιλαμβάνει πράκτορες και τις αντίστοιχες ενέργειές τους. Ο φάκελος των παραδειγμάτων δεν είναι προφανώς απαραίτητος για τη λειτουργία της βιβλιοθήκης αλλά αποτελεί ιδανική αρχή για τη μελέτη του τρόπου χρήσης της. Αναλυτικότερη περιγραφή των παραδειγμάτων ακολουθεί στις επόμενες ενότητες.



Εικόνα 6. Το δέντρο καταλόγων της βιβλιοθήκης

Επιπλέον, στο αποθετήριο κώδικα της βιβλιοθήκης (περισσότερες πληροφορίες στο Παράρτημα 1) βρίσκονται διαθέσιμα και project files και demos που καθιστούν ευκολότερη τη μελέτη και εκτέλεση των διαθέσιμων κόσμων που έχουν αναπτυχθεί.

3.4 Παραδείγματα βιβλιοθήκης iThink

Στα πλαίσια της πτυχιακής μας εργασίας μελετήθηκαν δύο πεδία σχεδιασμού και περιγραφής κόσμων, το BlocksWorld και το SimpleFPS_lite.

Το BlocksWorld είναι το κλασικό παράδειγμα κόσμου για μελέτη σχεδιαστών, που περιγράφει ένα κόσμο κύβων που οργανώνονται πάνω σε μια επιφάνεια. Κατά συνέπεια, μια αρχική κατανομή αυτών αποτελεί την αρχική κατάσταση και ο στόχος είναι μια επιθυμητή κατανομή τους.

Ο κόσμος SimpleFPS [10] αποτελεί μια απλοποιημένη μεταφορά ενός τυπικού περιβάλλοντος παιχνιδιού δράσης πρώτου προσώπου, όπου οι πράκτορες αξιοποιούν το περιβάλλον τους και αντικείμενα ώστε να υπερνικήσουν τους ανταγωνιστές τους. Στην εργασία μας έχουμε υλοποιήσει ένα υποσύνολο αυτής της περιγραφής, την οποία ονομάσαμε SimpleFPS_lite.

3.4.1 BlocksWorld

```
public class BlocksWorldAgent : MonoBehaviour
{
   iThinkBrain brain;
   public string[] schemaList = {
                                 "ActionBwPickUp-1-Tag~block",
                                 "ActionBwUnStack-2-Tag~block-Tag~block",
                                 "ActionBwPutDown-1-Tag~block",
                                 "ActionBwStack-2-Tag~block-Tag~block",
                               };

   public void Awake()
   {
      brain = new iThinkBrain();

      // Sensing GameObjects
      List<String> tags = new List<String>();
      tags.Add( "block" );
      brain.sensorySystem.OmniscientUpdate( this.gameObject, tags );

      // Building knowledge
      initKnowledge();
      brain.curState = brain.startState;

      // Building Actions from knowledge
      brain.ActionManager = new iThinkActionManager();
      brain.ActionManager.initActionList(    this.gameObject, brain.getKnownObjects(),
                                             schemaList, brain.getKnownFacts() );

      // Init search
      brain.planner.forwardSearch(  brain.startState, brain.goalState, brain.ActionManager );
   }
}
```

Σχήμα 1. Στοιχειώδης υλοποίηση του BlocksWorld πράκτορα

3.4.2 SimpleFPS_lite

```
public class SimpleGameAgent : MonoBehaviour
{
    iThinkBrain brain;

    public string[] schemaList = {
                        "ActionSGMove-3-Tag~location-Tag~location-Tag~direction",
                        "ActionSGTurn-2-Tag~direction-Tag~direction",
                        "ActionSGShoot-4-Tag~location-Tag~location-Tag~direction-Tag~gun",
                        "ActionSGStab-2-Tag~location-Tag~knife",
                        "ActionSGPickUp-2-Tag~knife-Tag~location",
                        "ActionSGPickUp-2-Tag~gun-Tag~location"
                       };

    public void Awake()
    {
        brain = new iThinkBrain();

        // Sensing GameObjects
        List<String> tags = new List<String>();
        tags.Add( "direction" ); tags.Add( "location" ); tags.Add( "player" );
        tags.Add( "npc" ); tags.Add( "gun" ); tags.Add( "knife" );
        brain.sensorySystem.OmniscientUpdate( this.gameObject, tags );

        Debug.LogWarning( "knownObjects count: " +
brain.sensorySystem.getKnownObjects().Count );

        // Building knowledge
        SimplegameTest();
        brain.curState = brain.startState;

        // Building Actions from knowledge
        brain.ActionManager = new iThinkActionManager();
        brain.ActionManager.initActionList( this.gameObject, schemaList,
brain.getKnownObjects(), brain.getKnownFacts() );

        Debug.LogWarning( "action count: " + brain.ActionManager.getActions().Count );

        // Init search
        brain.planner.forwardSearch( brain.startState, brain.goalState,
brain.ActionManager);
        brain.planner.getPlan().debugPrintPlan();
    }
```

Σχήμα 2. Στοιχειώδης υλοποίηση του SimpleFPS_lite πράκτορα

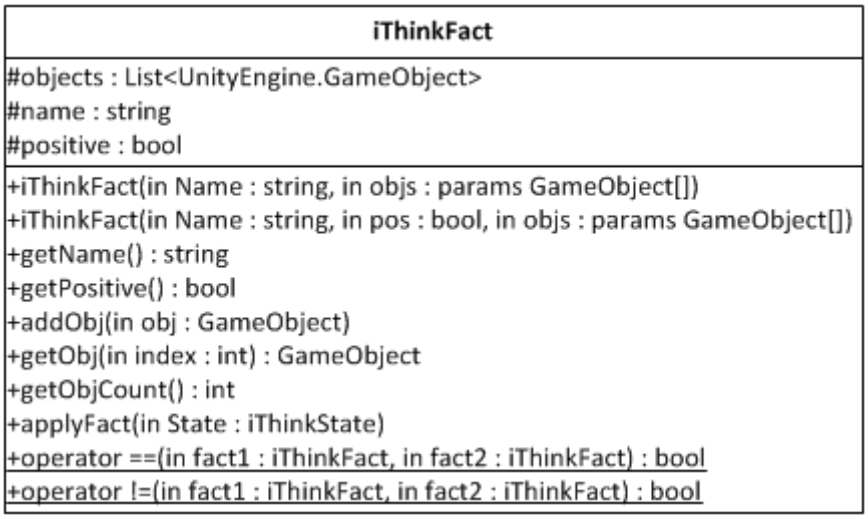
3.5 Περιγραφή κλάσεων βιβλιοθήκης iThink

Σε αυτό το κεφάλαιο μελετήσαμε επιγραμματικά τη λειτουργία της βιβλιοθήκης και την οργάνωση του πηγαίου κώδικα ενώ είδαμε παραδείγματα χρήσης της βιβλιοθήκης μέσω της υλοποίησης πρακτόρων. Μπορούμε πλέον να μελετήσουμε αναλυτικότερα την περιγραφή, τη χρήση και την υλοποίηση των κλάσεων της βιβλιοθήκης.

Για κάθε μια από τις εννέα κλάσεις θα παρουσιαστεί μια επιγραμματική περιγραφή της, θα δοθούν παραδείγματα αρχικοποίησης και χρήσης, και τέλος θα αναλυθεί η υλοποίησή τους.

Εφόσον είναι επιθυμητή η ανάπτυξη/βελτίωση της βιβλιοθήκης, παρέχεται και αναλυτική τεχνική τεκμηρίωση στο σχετικό παράρτημα της παρούσας εργασίας, την οποία καλείται ο χρήστης να μελετήσει.

3.5.1 iThinkFact



Εικόνα 7. UML διάγραμμα κλάσης iThinkFact

Γενική περιγραφή

Αποτελεί τη περιγραφή ενός STRIPS γεγονότος και χρησιμοποιείται στη περιγραφή καταστάσεων και ενεργειών.

Παράδειγμα αρχικοποίησης και χρήσης

```
class ActionBwPickUp : iThinkAction
{
    public override void initEffects()
    {
        effects.Add( new iThinkFact( "clear", false, Obj ) );
        effects.Add( new iThinkFact( "onTable", false, Obj ) );
        effects.Add( new iThinkFact( "gripEmpty", false ) );
        effects.Add( new iThinkFact( "holding", Obj ) );
    }
}
```

Αναλυτική περιγραφή

Ένα iThinkFact χρησιμοποιείται στην iThink στις εξής περιπτώσεις:

- στη περιγραφή των προϋποθέσεων και των επιδράσεων μιας ενέργειας,
- στη περιγραφή μιας κατάστασης, και
- στη συλλογή γνώσης για τη δημιουργία εφαρμόσιμων ενεργειών

Ο χρήστης της iThink θα ασχοληθεί άμεσα μόνο με τις δύο πρώτες χρήσεις, ενώ η τρίτη πραγματοποιείται στην εσωτερική λειτουργία της κλάσης iThinkActionSchemas. Η

δημιουργία ενός γεγονότος αντιστοιχεί απλά στην ορθή κλήση της συνάρτησης δημιουργίας (constructor) όπως ορίζει το πρότυπο (εικόνα 7) και στην εισαγωγή του στη λίστα που ενδιαφέρει το χρήστη.

Περιγραφή υλοποίησης

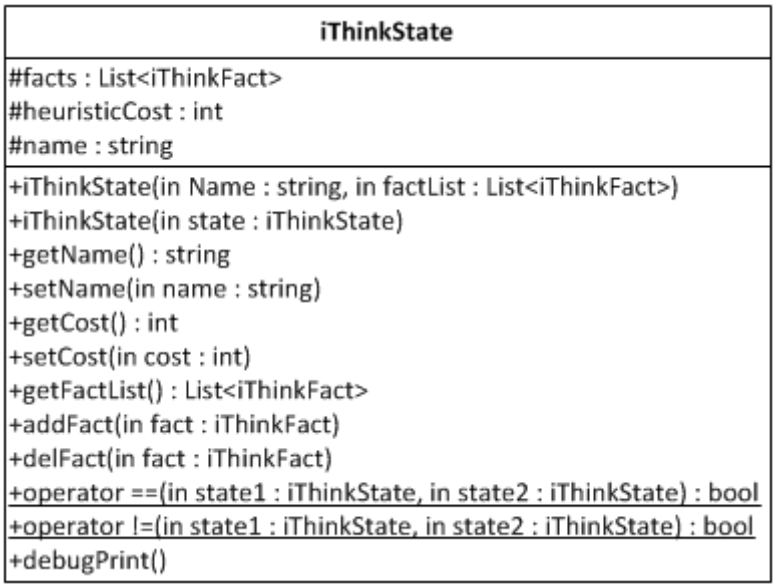
Η κλάση iThinkFact χρησιμοποιείται για τη δημιουργία λογικών λεκτικών, καθώς ένα γεγονός της βιβλιοθήκης iThink αντιστοιχεί σε ένα λεκτικό της αναπαράστασης STRIPS του κόσμου που περιγράφεται. Κατά συνέπεια, χρειαζόμαστε:

- το όνομα του κατηγορήματος (string name)
- μια δυαδική μεταβλητή που δηλώνει αν το λεκτικό είναι θετικό ή όχι (bool positive)
- μια λίστα από GameObjects, τα οποία αντιστοιχούν σε στιγμιότυπα του κόσμου και αποτελούν τα ορίσματα του σχετικού κατηγορήματος (List<GameObject> objects)

Πλέον μπορούμε να δημιουργήσουμε οποιοδήποτε λεκτικό, αρκεί να δημιουργήσουμε ένα αντικείμενο τύπου iThinkFact.

Για να γίνει πιο εύχρηστη η κλάση, έγινε υπερφόρτωση των τελεστών == και != ούτως ώστε να εκτελούν σύγκριση των αντίστοιχων γεγονότων. Επιπλέον, δημιουργήθηκε η συνάρτηση applyFact(), η οποία εφαρμόζει το γεγονός στη κατάσταση-παράμετρο τύπου iThinkState. Έτσι, ένα θετικό λεκτικό προστίθεται στη κατάσταση, ενώ ένα αρνητικό αφαιρείται, λειτουργία που αντιστοιχεί στη κατάστρωση σχεδίου κατά STRIPS.

3.5.2 iThinkState



Εικόνα 8. UML διάγραμμα κλάσης iThinkState

Γενική περιγραφή

Αποτελεί τη περιγραφή ενός κόμβου-κατάστασης του χώρου αναζήτησης.

Παράδειγμα αρχικοποίησης και χρήσης

1) Δημιουργία αρχικής κατάστασης στη κατάστρωση σχεδίου με βάση τη γνώση του χρήστη (η οποία διατηρείται στη λίστα factList):

```
brain.startState = new iThinkState( "Initial", new List<iThinkFact>( factList ) );
```

2) Ενδεικτική συνάρτηση εφαρμογής της αμέσως επόμενης ενέργειας ενός πλάνου μέσω της μετάβασης σε νέα κατάσταση με βάση την τωρινή:

```
void applyNextAction()
{
    List<iThinkAction> actionList = brain.planner.getPlan().getPlanActions();
    brain.curState = new iThinkState( actionList[0].applyEffects( brain.curState ) );
    actionList.RemoveAt( 0 );
}
```

Αναλυτική περιγραφή

Η iThinkState αποτελεί ουσιαστικά μια συλλογή γεγονότων σε συνδυασμό, προαιρετικά, με το συνολικό κόστος της κατάστρωσης σχεδίου μέχρι τον αντίστοιχο κόμβο. Ο χρήστης αναλαμβάνει να ορίσει την αρχική κατάσταση και την κατάσταση στόχου, τις οποίες αξιοποιεί ο σχεδιαστής για να καταστρώσει σχέδιο ενεργειών. Οποιαδήποτε ενδιάμεση κατάσταση προφανώς αποτελεί και αυτή iThinkState.

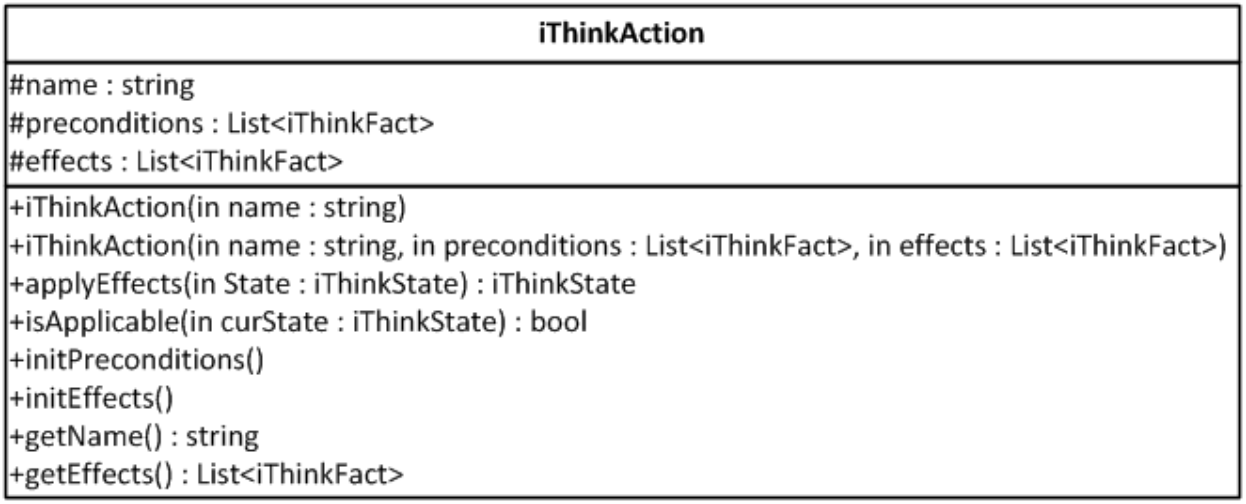
Περιγραφή υλοποίησης

Η κλάση iThinkState αντιπροσωπεύει μια κατάσταση του χώρου αναζήτησης και αποτελεί ουσιαστικά μια συλλογή από γεγονότα. Για τη περιγραφή της χρειαζόμαστε:

- Μια λίστα από γεγονότα (`List<iThinkFact> facts`),
- προαιρετικά, το κόστος αναζήτησης μέχρι την αντίστοιχη κατάσταση-κόμβο, εφόσον χρειάζεται για τον αλγόριθμο αναζήτησης (`int heuristicCost`)
- προαιρετικά, ένα όνομα για την ευκολότερη εσωτερική αναπαράσταση και αποσφαλμάτωση της κατάστρωσης σχεδίου (`string name`)

Από κει και πέρα, χρειαζόμαστε συναρτήσεις για τη προσθήκη (`addFact()`) και αφαίρεση (`delFact()`) γεγονότων από τη κατάσταση.

3.5.3 iThinkAction



Εικόνα 9. UML διάγραμμα κλάσης iThinkAction

Γενική περιγραφή

Αποτελεί τη περιγραφή μιας υλοποιημένης ενέργειας.

Παράδειγμα αρχικοποίησης και χρήσης

1) Ορισμός συνάρτησης “Pick-Up” στον κόσμο του BlocksWorld. Ο χρήστης αρχικοποιεί τις προϋποθέσεις και τις επιδράσεις της ενέργειας και διαχειρίζεται τα σχετικά GameObjects (στη παρούσα περίπτωση, το αντικείμενο που πρόκειται να σηκωθεί):

```
class ActionBwPickUp : iThinkAction
{
    GameObject Obj;
    public ActionBwPickUp( string name, GameObject obj )
        : base( name )
    {
        Obj = obj;

        initPreconditions();
        initEffects();
    }

    public override void initPreconditions()
    {
        preConditions.Add( new iThinkFact( "clear", Obj ) );
        preConditions.Add( new iThinkFact( "onTable", Obj ) );
        preConditions.Add( new iThinkFact( "gripEmpty" ) );
```

```
    }

    public override void initEffects()
    {
        effects.Add( new iThinkFact( "clear", false, Obj ) );
        effects.Add( new iThinkFact( "onTable", false, Obj ) );
        effects.Add( new iThinkFact( "gripEmpty", false ) );
        effects.Add( new iThinkFact( "holding", Obj ) );
    }
}
```

2) Ο χρήστης αναλαμβάνει να ορίσει τον τρόπο δημιουργίας των αντικειμένων της αντίστοιχης ενέργειας στη συνάρτηση iThinkActionSchemas::actionGenerator(), ούτως ώστε να μπορούν να δημιουργηθούν οι εφαρμόσιμες ενέργειες με βάση τη γνώση του πράκτορα και να γίνουν διαθέσιμες μέσω του iThinkActionManager:

iThinkActionSchemas.cs:

```
    public virtual void actionGenerator( List<GameObject[]> combinations )
    {
        iThinkAction action;
        foreach ( GameObject[] matrix in combinations )
        {
            switch ( schemaElements[0] )
            {
                case "ActionBwUnStack":
                    action = new ActionBwUnStack( "UnStack", matrix[0], matrix[1] );
                    tempActionList.Add( action );
                    break;
            }
        }
    }
```

BlocksWorldAgent.cs:

```
    public string[] schemaList = { "ActionBWPickUp-1-Tag~block" };
//...
    brain.ActionManager.initActionList( this.gameObject, brain.getKnownObjects(),
                                        schemaList, brain.getKnownFacts() );
```

Αναλυτική περιγραφή

Ο χρήστης πρέπει αρχικά να περιγράψει τη μορφή των ενεργειών του πράκτορα (το λεγόμενο “action schema”), ούτως ώστε να είναι δυνατή η αυτόματη δημιουργία αυτών από την iThinkActionManager. Η παραπάνω διαδικασία αναλύεται στην υποενότητα της iThinkActionSchemas.

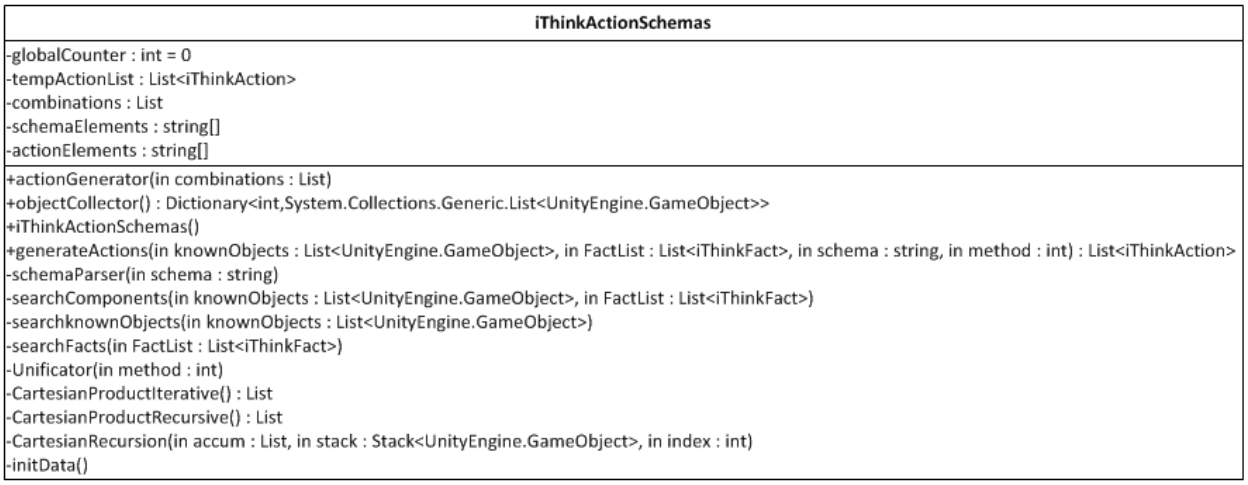
Εφόσον ο χρήστης προσθέσει προϋποθέσεις και επιδράσεις, μένει να προσθέσει τον κώδικα αρχικοποίησης νέων αντικειμένων της ενέργειας στη συνάρτηση iThinkActionSchemas::actionGenerator(), ούτως ώστε να μπορεί μετά η iThinkActionManager να δημιουργεί εφαρμόσιμες ενέργειες.

Η βιβλιοθήκη δεν ασχολείται με την εφαρμογή των ενεργειών στο περιβάλλον της Unity3D, καθώς αυτή είναι ευθύνη του χρήστη, επομένως περιορίζεται μόνο στη διαχείριση αυτών και στη κατάστρωση σχεδίου.

Περιγραφή υλοποίησης

Λόγω της αυστηρότητας τύπων της γλώσσας C#, όπου χρειαζόμασταν συλλογή ενεργειών έπρεπε να έχουμε έναν τύπο, τον iThinkAction. Κατά συνέπεια, κάθε ενέργεια του πράκτορα πρέπει να οριστεί ως θυγατρική κλάση αυτής, δίνοντάς μας τη δυνατότητα να οργανώσουμε κάθε ενέργεια ξεχωριστά με τα δικά της απαραίτητα δεδομένα. Λειτουργίες κοινές για όλες τις ενέργειες, όπως ο έλεγχος εφαρμοσιμότητας και η εφαρμογή των επιδράσεων της ενέργειας σε μια κατάσταση, προστίθενται στη κλάση iThinkAction.

3.5.4 iThinkActionSchemas



Εικόνα 10. UML διάγραμμα κλάσης iThinkActionSchemas

Γενική περιγραφή

Συλλέγει και αναγνωρίζει τις περιγραφές των ενεργειών με βάση ένα συγκεκριμένο format και, αξιοποιώντας τη γνώση του πράκτορα, παράγει τις διαθέσιμες για αυτόν ενέργειες.

Παράδειγμα αρχικοποίησης και χρήσης

Δεν υπάρχει διεπαφή με το χρήστη όσον αφορά την iThinkActionSchemas. Ο χρήστης αναλαμβάνει μόνο να περιγράψει τα schemas των actions του πράκτορα:

BlocksWorldAgent.cs:

```
    public string[] schemaList = { "ActionBWPickUp-1-Tag~block" };
```

και έπειτα να επεξεργαστεί τη συνάρτηση iThinkActionSchemas::generateActions() όπως θα αναλυθεί ευθύς αμέσως.

Αναλυτική περιγραφή/Περιγραφή υλοποίησης

Ο τρόπος λειτουργίας της κλάσης εμφανίζεται συνοπτικά στη συνάρτηση iThinkActionSchemas::generateActions():

```
    public List<iThinkAction> generateActions( List<GameObject> knownObjects,
List<iThinkFact> FactList, string schema, int method )
    {
        initData();
        schemaParser( schema );
        searchComponents( knownObjects, FactList );
```

```
        Unificator( method );
        actionGenerator( combinations );
        return tempActionList;
    }
```

Για κάθε συμβολοσειρά περιγραφής ενέργειας (action schema) του πράκτορα, καλείται στον iThinkActionManager η συνάρτηση generateActions(). Αφού αρχικοποιηθούν τα εσωτερικά δεδομένα της κλάσης iThinkActionSchemas, χωρίζεται η συμβολοσειρά στα σύμβολα/μέρη (tokens) της. Έπειτα, με βάση τη γνώση του πράκτορα για τον κόσμο του (δηλαδή το σύνολο των γνωστών αντικειμένων τύπου GameObject και των γεγονότων τύπου iThinkFact) μαζεύονται όλα τα σχετικά στοιχεία για τη δημιουργία της ενέργειας. Αφού ολοκληρωθεί η συλλογή τους, πρέπει να δημιουργηθούν οι σχετικές ομάδες αυτών, η καθεμία από τις οποίες θα αποτελεί το σύνολο ορισμάτων τις εκάστοτε ενέργειας.

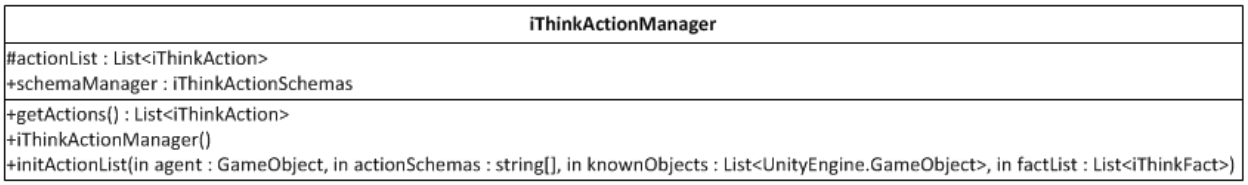
Αφού παραχθούν όλοι οι συνδυασμοί, αναλαμβάνει η συνάρτηση actionGenerator() να παράγει όλες τις εφαρμόσιμες ενέργειες του πράκτορα. Αυτή η συνάρτηση παρουσιάζει ενδιαφέρον για το χρήστη, αφού είναι η μόνη που αναλαμβάνει να επεξεργαστεί για το εκάστοτε πρόβλημα/περιβάλλον. Έτσι, για παράδειγμα, για να μπορεί η iThinkActionSchemas να παράγει ενέργειες τύπου “ActionBwUnStack”, χρειάζεται ο χρήστης να γράψει:

```
        case "ActionBwUnStack":
            action = new ActionBwUnStack( "UnStack", matrix[0], matrix[1] );
            tempActionList.Add( action );
            break;
```

Η συμβολοσειρά στο keyword “case” αντιστοιχεί στο πρώτο σύμβολο της σχετικής περιγραφής ενέργειας, που δηλώνει πως επιθυμούμε να παράγουμε ενέργεια τύπου ActionBwUnStack. Επομένως, δημιουργούμε και αρχικοποιούμε τη σχετική ενέργεια στη μεταβλητή “action”, την οποία προσθέτουμε και στη λίστα διαθέσιμων ενεργειών που παράγουμε. Για τη δημιουργία της ενέργειας, ακολουθούμε το πρότυπο της συνάρτησης, και επομένως αναλαμβάνουμε την εισαγωγή όσων αντικειμένων της αντιστοιχούν (matrix[0] και matrix[1] στη συγκεκριμένη περίπτωση).

Η αυτοματοποίηση αυτής της διαδικασίας απαιτεί εναλλακτικές λύσεις όπως η χρήση συγκεκριμένων μοτίβων σχεδίασης, τις οποίες προτιμήσαμε να αποφύγουμε σε αυτή τη φάση της συγγραφής του κώδικα, αλλά που θα συμπεριλάβουμε σε μελλοντική έκδοσή του.

3.5.5 iThinkActionManager



Εικόνα 11. UML διάγραμμα κλάσης iThinkActionManager

Γενική περιγραφή

Καλεί τη συνάρτηση δημιουργίας διαθέσιμων ενεργειών της iThinkActionSchemas και τις συλλέγει για εύκολη πρόσβαση.

Παράδειγμα αρχικοποίησης και χρήσης

1) Ο χρήστης αναλαμβάνει μόνο να περιγράψει τα schemas των ενεργειών, να συλλέξει γνώση για τον πράκτορα (μέσω για παράδειγμα του iThinkSensorySystem) και από κει και πέρα αναλαμβάνει ο iThinkActionManager τη δημιουργία των διαθέσιμων ενεργειών.

BlocksWorldAgent.cs:

```
      public string[] schemaList = { "ActionBwPickUp-1-Tag~block" };
//...
      // Sensing GameObjects
      List<String> tags = new List<String>();
      tags.Add( "block" );
      brain.sensorySystem.OmniscientUpdate( this.gameObject, tags );
//...
      // Generate actions from knowledge
      brain.ActionManager = new iThinkActionManager();
      brain.ActionManager.initActionList(   this.gameObject, brain.getKnownObjects(),
                                            schemaList, brain.getKnownFacts() );
```

Αναλυτική περιγραφή

Παρέχοντας το αντικείμενο που αναπαριστά τον πράκτορα, τη λίστα γνωστών αντικειμένων και τη λίστα των γνωστών γεγονότων, η iThinkActionManager::initActionList() παράγει τη λίστα των διαθέσιμων ενεργειών, ένα υποσύνολο των οποίων είναι εφαρμόσιμο σε οποιαδήποτε φάση της αναζήτησης.

Περιγραφή υλοποίησης

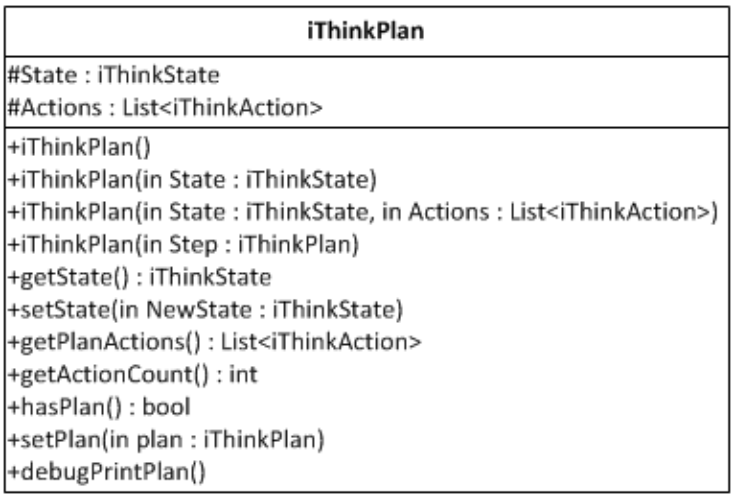
```
    /// This function will be called by the user whenever he wants to generate all
actions available to him.
    public void initActionList( GameObject agent, string[] actionSchemas,
List<GameObject> knownObjects, List<iThinkFact> factList )
    {
        List<iThinkAction> tempActionList = new List<iThinkAction>();
        foreach ( string schema in actionSchemas )
        {
            tempActionList.Clear();
            tempActionList = schemaManager.generateActions( knownObjects, factList,
schema, 2 );
            actionList.AddRange( tempActionList );
        }
    }
```

Όλη η ουσιαστική δουλειά γίνεται στη γραμμή:

```
tempActionList = schemaManager.generateActions( knownObjects, factList, schema, 2 );
```

όπου ζητείται από την iThinkActionSchemas να παράγει τις διαθέσιμες ενέργειες με βάση τις πληροφορίες που παρέχονται.

3.5.6 iThinkPlan



Εικόνα 12. UML διάγραμμα κλάσης iThinkPlan

Γενική περιγραφή

Αποτελεί απλοποίηση της υλοποίησης και χρήσης του καταστρωμένου σχεδίου.

Παράδειγμα αρχικοποίησης και χρήσης

Ο χρήστης δεν αναλαμβάνει άμεσα τη δημιουργία της iThinkPlan, αφού αυτή παράγεται μόλις ο χρήστης εκτελέσει μια αναζήτηση μέσω του iThinkPlanner. Έπειτα, απλά αξιοποιεί τη λίστα των ενεργειών που αποτελούν το σχέδιο:

BlocksWorldAgent.cs:

```
        brain.planner.forwardSearch(  brain.startState, brain.goalState,
                                      brain.ActionManager, 1 );
//...
        if ( brain.planner.getPlan().hasPlan() )
           applyNextAction();    // Υποθετική συνάρτηση που εφαρμόζει την επόμενη
                                 // ενέργεια του σχεδίου
```

Αναλυτική περιγραφή

Η iThinkPlan φυλάσσει ένα ζεύγος πληροφορίας: το σύνολο των ενεργειών που έχει οριστεί να εκτελεστεί με βάση την επίτευξη ορισμένου στόχου, καθώς και τη κατάσταση στην οποία θα οδηγήσει η ακολουθία αυτή εφόσον εφαρμοστεί στην αρχική κατάσταση. Χρησιμοποιείται και για την αναπαράσταση ενδιάμεσων βημάτων στη κατάστρωση σχεδίου, με το τελικό βήμα αυτής να αποτελεί και το τελικό σχέδιο. Ο χρήστης ενδιαφέρεται για τη συνάρτηση hasPlan(), που ελέγχει εάν υπάρχουν διαθέσιμες ενέργειες προς εκτέλεση.

3.5.7 iThinkPlanner

iThinkPlanner
#Plan : iThinkPlan #_OpenStates : List<iThinkPlan> #_VisitedStates : List<iThinkState>
+iThinkPlanner() +getPlan() : iThinkPlan +forwardSearch(in InitialState : iThinkState, in GoalState : iThinkState, in ActionManager : iThinkActionManager) : bool +forwardSearch(in InitialState : iThinkState, in GoalState : iThinkState, in ActionManager : iThinkActionManager, in Method : int) : bool +depthFS(in GoalState : iThinkState, in ActionManager : iThinkActionManager, in OpenStates : List<iThinkPlan>, in VisitedStates : List<iThinkState>) : iThinkPlan +breadthFS(in GoalState : iThinkState, in ActionManager : iThinkActionManager, in OpenStates : List<iThinkPlan>, in VisitedStates : List<iThinkState>) : iThinkPlan +bestFS(in GoalState : iThinkState, in ActionManager : iThinkActionManager, in OpenStates : List<iThinkPlan>, in VisitedStates : List<iThinkState>) : iThinkPlan -hFunction(in nextState : iThinkState, in GoalState : iThinkState) : int +progress(in Step : iThinkPlan, in Action : iThinkAction) : iThinkPlan +getApplicable(in State : iThinkState, in Actions : List<iThinkAction>) : List<iThinkAction> -compareStates(in curState : iThinkState, in goalState : iThinkState) : bool

Εικόνα 13. UML διάγραμμα κλάσης iThinkPlanner

Γενική περιγραφή

Υλοποιεί το σχεδιαστή και εκτελεί τη κατάστρωση σχεδίου με διαφορετικούς τρόπους.

Παράδειγμα αρχικοποίησης και χρήσης

1) Εφόσον ο χρήστης αξιοποιήσει στον πράκτορά του τη κλάση iThinkBrain, δε χρειάζεται να αρχικοποιήσει τον planner. Αν θέλει να χρησιμοποιήσει μόνο τον iThinkPlanner, τότε αρκεί μια αρχικοποίηση:

```
planner = new iThinkPlanner();
```

2) Για να εκτελέσει ο σχεδιαστής την αναζήτηση, χρειάζεται τρεις παραμέτρους:

- Αρχική κατάσταση
- Κατάσταση στόχου
- Έναν αρχικοποιημένο iThinkActionManager που περιέχει εφαρμόσιμες ενέργειες, όπως δείξαμε στις υποενότητες των κλάσεων iThinkActionSchemas και iThinkActionManager.
- Μια προαιρετική 4^{η} παράμετρο που υποδηλώνει τον τύπο της αναζήτησης, δηλαδή ποιος αλγόριθμος αναζήτησης θα εκτελεστεί.

```
brain.planner.forwardSearch( brain.startState, brain.goalState, brain.ActionManager);
```

3) Από εκεί και πέρα, έχοντας εκτελέσει την κατάστρωση σχεδίου, ο χρήστης μπορεί να λάβει το παραγόμενο αντικείμενο τύπου iThinkPlan μέσω της iThinkPlanner::getPlan() και να δράσει ανάλογα.

Αναλυτική περιγραφή

Ο σχεδιαστής χρειάζεται να ελέγχει αν η τωρινή κατάσταση είναι κατάσταση στόχου, λειτουργία που καλύπτει η συνάρτηση compareStates().

Σε κάθε βήμα της αναζήτησης, χρειάζεται να γνωρίζουμε το σύνολο των εφαρμόσιμων ενεργειών, το οποίο λαμβάνουμε με τη συνάρτηση getApplicable().

Για τη μετάβαση σε επόμενη κατάσταση, χρησιμοποιείται η συνάρτηση progress(), η οποία παράγει ένα επόμενο βήμα στη κατάστρωση σχεδίου μέσω της εφαρμογής μιας εφαρμόσιμης ενέργειας στη τωρινή κατάσταση και συνδυασμού αυτής με την αντίστοιχη ακολουθία ενεργειών, δημιουργώντας πρακτικά ένα ημιτελές πλάνο.

Μόλις βρεθεί βήμα της κατάστρωσης σχεδίου που περιγράφει κατάσταση στόχου σημαίνει πως βρέθηκε πλάνο ενεργειών για την εκπλήρωση του στόχου, το οποίο και επιστρέφεται από τη συνάρτηση forwardSearch(). Αν δε βρεθεί τέτοιο πλάνο, επιστρέφεται κενός δείκτης.

Περιγραφή υλοποίησης

Ο σχεδιαστής εκτελεί εμπρόσθια αναζήτηση (forward search) ακολουθώντας τον παρακάτω αλγόριθμο:

ForwardSearch

```
OpenStates.Insert(S1);
VisitedStates.Insert(S1);

while ( OpenStates.NotEmpty() )
{
      Scur = OpenStates.GetNext();
      If ( Scur.IsGoal() )
            return true;
      foreach ( action in ApplicableActions(Scur))
      {
            Snext = ApplyEffects(Scur, action);
            if ( VisitedStates.Exists(Snext) )
                  continue;
            else
            {
                  VisitedStates.Insert(Snext);
                  OpenStates.Insert(Snext);
            }
       }
}

return false;
```

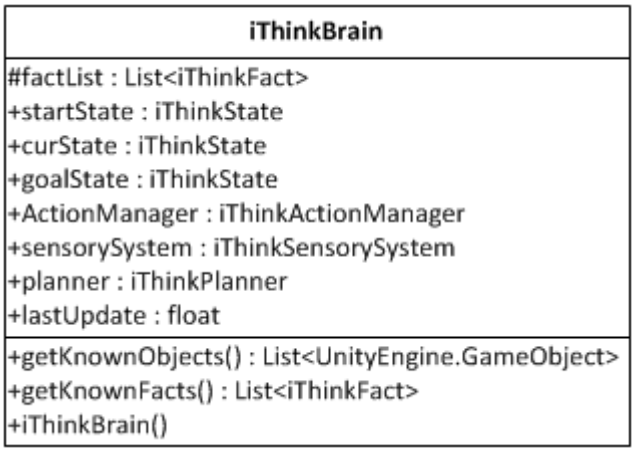
Σχήμα 3. Υλοποιημένος αλγόριθμος προς-τα-εμπρός αναζήτησης

Ενδιαφέρον παρουσιάζουν οι συλλογές OpenStates και VisitedStates, οι οποίες αντιστοιχούν στο σύνορο αναζήτησης (search fringe) και στο σύνολο των καταστάσεων που έχουν επεκταθεί ήδη. Η μεν OpenStates περιέχει το σύνολο των καταστάσεων από τις οποίες ο αλγόριθμος θα λαμβάνει την επόμενη κατάσταση προς μελέτη, ενώ η VisitedStates χρησιμοποιείται προς αποφυγή εισαγωγής ήδη μελετημένης γνώσης.

Από κει και πέρα, οι διαφορετικές μέθοδοι αναζήτησης αποτελούν διαφορετικές εκδοχές αυτής της βάσης, αλλάζοντας το πως εισάγονται τα ενδιάμεσα βήματα του σχεδιαστή στη λίστα OpenStates (σημειωμένη κόκκινη γραμμή). Για παράδειγμα, στην αναζήτηση κατά βάθος (DepthFS) τα νέα βήματα εισάγονται στην αρχή της λίστας, στην αναζήτηση κατά πλάτος (BreadthFS) τα νέα βήματα εισάγονται στο τέλος της λίστας, ενώ στην αναζήτηση κατά βέλτιστο κόστος (BestFS) όλα τα βήματα ταξινομούνται σε αύξουσα σειρά κόστους ώστε να επιλέγεται από την αρχή της λίστας το «βέλτιστο» βήμα με βάση την ευρετική συνάρτηση (απόσταση Manhattan).

Ο σχεδιαστής μπορεί εύκολα να επεκταθεί για να υλοποιηθεί ανάστροφη (backward) ή αμφικατευθυντήρια (bidirectional) αναζήτηση, ή να χρησιμοποιηθεί άλλη μέθοδος αναζήτησης όπως για παράδειγμα A*.

3.5.8 iThinkBrain



Εικόνα 14. UML διάγραμμα κλάσης iThinkBrain

Γενική περιγραφή

Αποτελεί εστιακό σημείο της χρήσης της βιβλιοθήκης iThink.

Παράδειγμα αρχικοποίησης και χρήσης

Κάθε πράκτορας θα πρέπει να είναι υποκλάση της MonoBehaviour, αφού ουσιαστικά λειτουργεί ως script στο περιβάλλον της Unity3D. Από κει και πέρα, ο χρήστης χρειάζεται να προσθέσει μια μεταβλητή τύπου iThinkBrain και μπορεί να αξιοποιήσει όλα του τα υποσυστήματα. Παρέχεται μια ενδεικτική υλοποίηση του πράκτορα του BlocksWorld:

```
public class BlocksWorldAgent : MonoBehaviour
{
    iThinkBrain brain;

    public string[] schemaList = {
                                    "ActionBwPickUp-1-Tag~block",
                                    "ActionBwUnStack-2-Tag~block-Tag~block",
                                    "ActionBwPutDown-1-Tag~block",
                                    "ActionBwStack-2-Tag~block-Tag~block",
                                };
//...
        brain = new iThinkBrain();

        // Sensing GameObjects
        List<String> tags = new List<String>();
        tags.Add( "block" );
        brain.sensorySystem.OmniscientUpdate( this.gameObject, tags );
```

```
        // Building knowledge
        initKnowledge();
        brain.curState = brain.startState;

        // Generate actions from knowledge
        brain.ActionManager = new iThinkActionManager();
        brain.ActionManager.initActionList( this.gameObject, brain.getKnownObjects(),
                                            schemaList, brain.getKnownFacts() );
//...
       // Init search
       brain.planner.forwardSearch(brain.startState,brain.goalState,brain.ActionManager);
    }
```

Αναλυτική περιγραφή

Το iThinkBrain απλά συλλέγει όλη τη λειτουργικότητα της βιβλιοθήκης σε μια κλάση, έτσι ο χρήστης παρέχει στον πράκτορά του όλη την απαραίτητη λειτουργικότητα που απαιτείται για τη διαδικασία κατάστρωσης σχεδίου.

Ενδιαφέρον έχουν οι μεταβλητές:

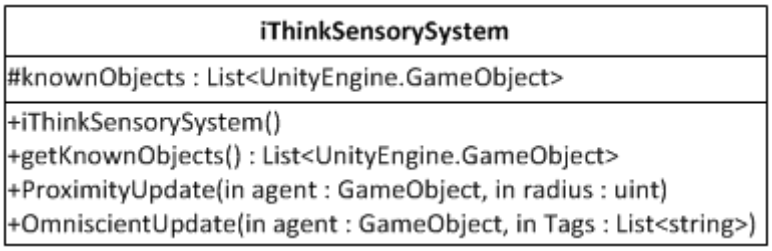
- ActionManager, τύπου iThinkActionManager
- sensorySystem, τύπου iThinkSensorySystem
- planner, τύπου iThinkPlanner
- factlist, που αποτελεί συλλογή όλων των facts που γνωρίζει ο πράκτορας

Στη παρούσα φάση, ο πράκτορας λειτουργεί με πλήρη γνώση, επομένως η factlist περιέχει όλα τα γεγονότα που είναι δυνατόν να γνωρίζει.

Μέσω των getKnownObjects() και getKnownFacts() μπορεί να λάβει ο χρήστης τις λίστες γνωστών αντικειμένων και γεγονότων του πράκτορα.

Τέλος, ο χρήστης καλείται να αρχικοποιεί τις μεταβλητές startState και goalState, που αντιστοιχούν στην αρχική κατάσταση και τη κατάσταση στόχου αντίστοιχα, ώστε να μπορέσει μετά ο σχεδιαστής να ξεκινήσει τη κατάστρωση σχεδίου.

3.5.9 iThinkSensorySystem



Εικόνα 15. UML διάγραμμα κλάσης iThinkSensorySystem

Γενική περιγραφή

Η χρήση της είναι προαιρετική αλλά απλοποιεί τη διαδικασία συλλογής γνώσης.

Παράδειγμα αρχικοποίησης και χρήσης

Η μεταβλητή sensorySystem του iThinkBrain αρχικοποιείται αυτόματα, οπότε δεν απαιτείται κάποια ιδιαίτερη ενέργεια από το χρήστη και μπορεί απευθείας να την αξιοποιήσει.

Έτσι, αν ο χρήστης θέλει να βρει όλα τα αντικείμενα με ετικέτα “block”, θα γράψει:

```
// Sensing GameObjects
List<String> tags = new List<String>();
tags.Add( "block" );
brain.sensorySystem.OmniscientUpdate( this.gameObject, tags );
```

Πλέον, μπορεί να αξιοποιήσει αυτά τα αντικείμενα με ευκολία σε αντικείμενο τύπου iThinkActionManager ούτως ώστε να παράγει εφαρμόσιμες ενέργειες σε αυτά με βάση τη περιγραφή των ενεργειών.

Αναλυτική περιγραφή/Περιγραφή υλοποίησης

Η κλάση iThinkSensorySystem παρέχει συναρτήσεις συλλογής γνώσης (όχι γεγονότων), δηλαδή αντικειμένων τύπου GameObject στο περιβάλλον της Unity3D που έχουν κατηγοριοποιηθεί με συγκεκριμένα Tags από το Περιβάλλον Εργασίας του εργαλείου.

Παρέχει δύο συναρτήσεις για εύρεση αντικειμένων, μια μέσω ετικετών (καθολική αναζήτηση) και μια μέσω εγγύτητας.

4. Προκλήσεις υλοποίησης σχεδιαστή σε μηχανή βιντεοπαιχνιδιών

4.1 Εισαγωγή

Όπως αναφέρθηκε και σε προηγούμενο κεφάλαιο η ιδέα της πτυχιακής ήταν η δημιουργία ενός γενικευμένου εργαλείου για το σχεδιασμό σε ηλεκτρονικά παιχνίδια. Προσπαθήσαμε να δημιουργήσουμε ένα πρόγραμμα το οποίο να προσεγγίζει ικανοποιητικά τις ακαδημαϊκές απαιτήσεις αλλά και να μην υστερεί από άποψη ευχρηστίας σε σχέση με έναν σχεδιαστή που προορίζεται αποκλειστικά για ηλεκτρονικά παιχνίδια.

4.2 Προκλήσεις Υλοποίησης

Αρχικός στόχος ήταν το εργαλείο που θα δημιουργήσουμε να είναι επαρκώς και πλήρως τεκμηριωμένο και σχετικά εύχρηστο, χωρίς να απαιτεί από το χρήστη πολλές γνώσεις προγραμματισμού. Θέλαμε να κάνουμε όσο το δυνατόν απλή και κατανοητή την οργάνωση του κώδικα για να είναι εύκολη η μελλοντική επέκταση του με επιπλέον εργαλεία και λειτουργίες.

Καθώς δεν υπάρχει μια τυπική μεθοδολογία για την υλοποίηση τεχνικών σχεδιασμού σε πλατφόρμες ανάπτυξης παιχνιδιών, μελετήσαμε τις διαθέσιμες υλοποιήσεις σχεδιαστών και αρκετές δημοσιεύσεις που αφορούσαν το θέμα της πτυχιακής μας. Τελικά, ύστερα από εκτεταμένη έρευνα, καταλήξαμε να μελετήσουμε σε βάθος τη μηχανή σχεδιασμού του παιχνιδιού F.E.A.R που είχε διαθέσιμο ολοκληρωμένο κώδικα και υλοποιούσε το σύστημα STRIPS (το οποίο υλοποιεί και περιγράφει εκτενώς η παρούσα πτυχιακή) , καθώς και τις δημοσιεύσεις που παρουσιάζονται στο κεφάλαιο των αναφορών.

Η μεγαλύτερη πρόκληση που αντιμετωπίσαμε στην αρχή ήταν η μελέτη του κώδικα του F.E.A.R. Ο κώδικας ήταν αρκετά καλά οργανωμένος αλλά υπήρχε ελλιπής, και σε πολλά σημεία μηδενική, τεκμηρίωση με αποτέλεσμα να είναι αδύνατο να κατανοήσουμε και να εκτιμήσουμε σωστά τις επιλογές αρχιτεκτονικής του τμήματος της τεχνητής νοημοσύνης. Όσον αφορά τη μελέτη των δημοσιεύσεων, ενώ παρουσίαζαν το θεωρητικό – ακαδημαϊκό κομμάτι του σχεδιασμού αρκετά καλά και ολοκληρωμένα, δεν υπήρχε επαρκής πληροφορία για το πώς μπορεί να εφαρμοστεί όλο αυτό το θεωρητικό υπόβαθρο στην πράξη. Η πρόκληση σε αυτό το κομμάτι της πτυχιακής ήταν η κλασική σε ζητήματα Μηχανικής Λογισμικού, ότι δηλαδή έπρεπε να γίνουν υποθέσεις για το σχεδιασμό χωρίς να είμαστε σίγουροι ότι έχουμε πάρει τις σωστές αποφάσεις. Λανθασμένες αποφάσεις και υποθέσεις θα μπορούσαν να οδηγήσουν σε σχεδιαστικά

αδιέξοδα, αύξηση της πολυπλοκότητας του προγράμματος, γενικές αδυναμίες, ανάγκη για επαναπροσδιορισμό των υποθέσεων ή και τελικά την αποτυχία των προβλεπόμενων στόχων, οδηγώντας σε αρκετές χαμένες ανθρωποώρες εργασίας.

Όταν η απαραίτητη έρευνα και μελέτη των κατάλληλων πηγών πραγματοποιήθηκαν και πάρθηκαν οι πρώτες σχεδιαστικές αποφάσεις και υποθέσεις, ξεκινήσαμε την υλοποίηση του εργαλείου. Σε αυτήν τη φάση της πτυχιακής είχαμε να αντιμετωπίσουμε αρκετές προκλήσεις και δύσκολα προβλήματα σχεδιασμού. Οι μεγαλύτερες προκλήσεις που αντιμετωπίσαμε στη φάση της υλοποίησης ήταν τα κενά και οι απορίες στη γνώση που είχαμε μελετήσει, τα οποία έπρεπε να κατανοήσουμε, να δούμε στην πράξη και να αντιμετωπίσουμε, καθώς και ορισμένα χαρακτηριστικά της γλώσσας προγραμματισμού C#, που όπως αποδείχτηκε σε αρκετές φάσεις περιέπλεκε τα πράγματα και δημιουργούσε επιπλέον προβλήματα.

Στη συνέχεια, θα παρουσιαστούν οι επιλογές σχεδιασμού που κάναμε στις διαφορετικές φάσεις υλοποίησης, τα προβλήματα που εμφανίστηκαν και ο τρόπος που τα αντιμετωπίσαμε από την αρχική μέχρι την τελική μορφή του κώδικα.

4.3 Προβλήματα κατά την υλοποίηση του iThink

Στις ακόλουθες ενότητες θα αναλυθούν προβλήματα και αδιέξοδα που συναντήσαμε κατά την υλοποίηση της βιβλιοθήκης. Δε θεωρούμε πως οι σχεδιαστικές επιλογές μας αποτελούν το μοναδικό ή ακόμα και το βέλτιστο τρόπο με τον οποίο μπορούσε να προσεγγιστεί το πρόβλημα της υλοποίησης σχεδιαστή. Θεωρούμε πως υπήρχαν ορισμένες εναλλακτικές επιλογές αρχιτεκτονικών και σχεδιαστικών επιλογών που μπορούσαμε να έχουμε ακολουθήσει, αλλά θα αναλύσουμε αυτές που τελικά υλοποιήσαμε και θεωρούμε πως ήταν οι καλύτερες για την εκπλήρωση των προσωπικών μας αναγκών αλλά και των στόχων που είχαμε θέσει.

4.3.1 Προβλήματα στην υλοποίηση του iThinkFact

4.3.1.1 Υλοποίηση του iThinkFact ακολουθώντας το F.E.A.R

Προσπάθεια

Αρχικά προσπαθήσαμε να προσεγγίσουμε τη λογική με την οποία η μηχανή σχεδιασμού του παιχνιδιού F.E.A.R υλοποιεί τα facts και να την προσαρμόσουμε στον κώδικά μας. Το F.E.A.R υλοποιεί τα γεγονότα του παιχνιδιού έχοντας έναν πίνακα από ακεραίους και κάθε μία θέση αυτού του πίνακα αντιπροσωπεύει ένα γεγονός το οποίο συμβαίνει στο παιχνίδι ή μία κατάσταση στην οποία μπορεί να βρεθεί ένας παίκτης. Με τη χρήση αυτής της απαρίθμησης επιτυγχάνουν να αντιστοιχιστεί μια σταθερά/ακέραιος σε κάθε λογική έκφραση και κατ' επέκταση σε κάθε γεγονότος (αφού στην υλοποίηση πρακτικά ταυτίζονται οι έννοιες). Ταυτόχρονα, οντότητες του παιχνιδιού όπως οι πράκτορες αντιστοιχούν και αυτές σε έναν κωδικό αριθμό/ακέραιο η καθεμία, με αποτέλεσμα να γίνεται εύκολη η δημιουργία των καταστάσεων του παιχνιδιού για κάθε επανάληψη του εργαλείου σχεδιασμού.

Ως συνέπεια των παραπάνω, ξεκινήσαμε να υλοποιούμε το iThinkFact ως κλάση που θα συμπεριλάμβανε διαφορετικές λογικές εκφράσεις. Κάθε μία από αυτές τις λογικές εκφράσεις θα έχει διαφορετικές παραμέτρους που κάθε μία αντιστοιχεί σε μία δυάδα πληροφορίας, η οποία περιγράφει τον τύπο της πληροφορίας και τη τιμή της. Ο «τύπος» αντιστοιχούσε στο είδος της οντότητας που αφορούσε το γεγονός (integer, float, GameObject, Component κ.α.) και αντιστοιχούσε στη απαρίθμηση iThinkPropType. Η τιμή του γεγονότος περιγραφόταν από αντικείμενο κλάσης iThinkPropValue που περιλάμβανε μόνο διαφορετικές συναρτήσεις δημιουργίας και είχε ως στόχο να διατηρεί την τιμή που περιέγραφε το iThinkPropType. Οι ίδιες οι λογικές εκφράσεις περιγραφόταν και αυτές με μια σταθερά που συμβολίζαμε με iThinkProp. Το iThinkProp ήταν μία αρίθμηση που περιέγραφε τις λογικές εκφράσεις τις οποίες θα χρησιμοποιούσαμε στο σχεδιασμό.

Σκέψεις και εκτιμώμενα πλεονεκτήματα

Υλοποιώντας τα γεγονότα με αυτόν τον τρόπο θεωρούσαμε πως θα καταφέρουμε να τα διαχειριστούμε πιο απλά και αποτελεσματικότερα. Είχαμε την πεποίθηση πως, αφού η περιγραφή των αντικειμένων ήταν αρκετά απλή (αν και πλήρης), οι πληροφορίες που χρειαζόμασταν θα ήταν εύκολα προσβάσιμες αφού θα ψάχναμε πάντα συγκεκριμένες οντότητες που θα αντιστοιχούσαν στον τύπο του γεγονότος που θα χρησιμοποιούσαμε

στο σχεδιασμό. Επίσης μία τέτοια περιγραφή θεωρούσαμε ότι θα ήταν πιο εύκολο να την κατανοήσει και να την αξιοποιήσει ένας μη έμπειρος χρήστης.

Θεωρούσαμε πως θα είχαμε τη δυνατότητα να αναγάγουμε την περιγραφή των διαφορετικών καταστάσεων, καθώς και των προϋποθέσεων και επιδράσεων των ενεργειών, σε περιγραφή που θα αφορούσε μόνο τέτοιου είδους λογικών εκφράσεων το οποίο εκτιμούσαμε πως θα μας διευκόλυνε ιδιαίτερα στην υλοποίηση του κώδικα αφού θα ήταν άμεσα και εύκολα διαχειρίσιμα. Είχαμε εκτιμήσει επιπλέον πως με την απλοποίηση της περιγραφής των καταστάσεων θα διευκολύναμε και τη συνολική κατανόηση του κώδικα και της μεθόδου STRIPS, το οποίο θα καθιστούσε το εργαλείο άμεσα αξιοποιήσιμο και από την ακαδημαϊκή κοινότητα για την εμβάθυνση στο σχεδιασμό σε σύγχρονες εφαρμογές.

Καταλήγοντας, επηρεασμένοι από το F.E.A.R θεωρούσαμε πως οποιαδήποτε πληροφορία ήταν αναγκαία για την εκτέλεση του προγράμματος και την ολοκλήρωση του σχεδιασμού το πρόγραμμα θα ήταν σε θέση να την αντλήσει στο χρόνο εκτέλεσης, γεγονός που θα εξυπηρετούσε την υλοποίηση και θα διευκόλυνε στη συνολική οργάνωση και διευθέτηση μελλοντικών προβλημάτων.

Προβλήματα και δυσκολίες στην υλοποίηση

Οι δυσκολίες που αντιμετωπίσαμε σε αυτή τη φάση της υλοποίησης ήταν αρκετές και προέκυψαν αρκετά προβλήματα, τα οποία σε εκείνη τη φάση συγγραφής του κώδικα δεν ήταν άμεσα διαχειρίσιμα και επιλύσιμα.

Από τα σημαντικότερα προβλήματα που αντιμετωπίσαμε στην αρχή της υλοποίησης είναι οι ιδιαιτερότητες της γλώσσας προγραμματισμού C# έναντι της γλώσσας προγραμματισμού C++. Η C#, προσεγγίζοντας το καθαρό μοντέλο αντικειμενοστραφή προγραμματισμού, δεν παρέχει για παράδειγμα τη συνηθισμένη λειτουργία δημιουργίας μακροεντολών ή ορισμού σταθερών του προεπεξεργαστή (preprocessor). Ο προεπεξεργαστής χρησιμοποιήθηκε εκτενώς στο F.E.A.R, το οποίο είναι υλοποιημένο σε C++, για την άμεση παραγωγή κώδικα συναρτήσεων, αναγκαίου για την εκτέλεση του προγράμματος και την εύρεση των απαιτούμενων αντικειμένων, χωρίς να απαιτείται η άμεση υλοποίησή τους από το χρήστη. Χρησιμοποιώντας τη γλώσσα προγραμματισμού C#, για την υλοποίηση του προγράμματός μας, έπρεπε τέτοιου είδους συναρτήσεις να υλοποιηθούν από το χρήστη ούτως ώστε να προχωρήσει η διαδικασία μεταγλώττισης. Αυτό αποτέλεσε τροχοπέδη για τη μορφή του κώδικα που

είχαμε αφού, κατ' επέκταση του προηγούμενου προβλήματος, δεν μπορούσαμε να αντιστοιχίσουμε εύκολα τις αριθμήσεις που είχαμε ορίσει με τα iThinkPropType και iThinkProp σε πραγματικά αντικείμενα του προγράμματος.

Επίσης το γεγονός ότι η C# δεν υποστηρίζει ένωση (union) για σύνθετους τύπους προξένησε περισσότερα προβλήματα αφού δεν είχαμε άμεσο τρόπο να ομαδοποιήσουμε αποδοτικά σε μία ενιαία δομή τις διαφορετικές τιμές που μπορούσαν να έχουν τα αντικείμενα που περιγράφονταν από το iThinkPropType. Η μη ομαδοποίηση των αντικειμένων οδήγησε στη δημιουργία πολλαπλών συναρτήσεων δημιουργίας στο πρόγραμμα, και πιο συγκεκριμένα στη κλάση iThinkPropValue, για την αποτελεσματική διαχείριση αυτών. Το πρόβλημα που παρουσιάστηκε εδώ είναι ότι δεν μπορούσαμε, σε κάθε περίπτωση, να προβλέψουμε τι αντικείμενο θα έπρεπε να δημιουργηθεί και να αποθηκευτεί προκειμένου να καλέσουμε την κατάλληλη συνάρτηση δημιουργίας αντικειμένου. Ακολουθήσαμε κάποιες κατευθύνσεις για την επίλυση των προβλημάτων αυτών αλλά οδήγησαν σε αρκετά δυσνόητη υλοποίηση με αποτέλεσμα να αποτύχουμε σε ένα από τους βασικούς στόχους που είχαμε θέσει.

Επιπρόσθετα με την περιγραφή των γεγονότων ως λογικές εκφράσεις, δημιουργήθηκε μία πληθώρα προβλημάτων που δυσκόλεψε την υλοποίηση και την αποτελεσματική οργάνωση του κώδικα. Ενώ η περιγραφή των διαφορετικών γεγονότων μπορούσε να γίνει εύκολα και αποτελεσματικά, η περιγραφή των προϋποθέσεων(preconditions) και των επιδράσεων(effects) των ενεργειών (actions) δεν μπορούσαν να γίνουν επιτυχώς αφού ο τρόπος που ήταν δομημένα τα γεγονότα δεν επέτρεπε άμεση σύνδεση των αντικειμένων του προγράμματος.

Στην υλοποίηση των καταστάσεων δημιουργήθηκαν παρόμοια προβλήματα. Το γεγονός ότι δεν είχαμε ενιαία δομή για τις τιμές των αντικειμένων, το iThinkPropValue, οδήγησε σε δυσκολίες στην υλοποίηση των συγκρίσεων των διαφορετικών γεγονότων μέσα σε μία κατάσταση, και κατ' επέκταση των ίδιων των καταστάσεων, αφού έπρεπε να διακρίνουμε πολλές περιπτώσεις για να καλύψουμε τις διάφορες τιμές που εξυπηρετούσε τόσο το iThinkPropValue όσο και το iThinkPropType. Στην υλοποίηση αρκετών συναρτήσεων του συνολικού προγράμματος αντιμετωπίσαμε δυσκολίες εξαιτίας του τρόπου οργάνωσης της γνώσης από τα διαφορετικά γεγονότα, αφού δεν ήταν λίγες οι φορές που χρειαζόμασταν πληροφορίες από αντικείμενα τις οποίες θα μπορούσαμε να αντλήσουμε μόνο αν είχαμε διαθέσιμα τα άμεσα εμπλεκόμενα αντικείμενα.

Τελειώνοντας, με τη μορφή των γεγονότων, που είχαμε υλοποιήσει, αντιληφθήκαμε ότι είχαμε χάσει τη σύνδεση των οντοτήτων (τύπου GameObject) με τις κατάλληλες πληροφορίες που μπορεί να υπήρχαν αποθηκευμένες σε ένα γεγονός. Χαρακτηριστικό παράδειγμα αποτελεί η ενέργεια Move όπου, ενώ είχαμε αποθηκεύσει σωστά το αρχικό Component που καθόριζε τη θέση του πράκτορα κατά την εκτέλεση του προγράμματος, δεν είχαμε την ικανότητα να συνδέσουμε το Component αυτό με το αντικείμενο που υπήρχε στο παιχνίδι με αποτέλεσμα να μην μπορούμε να διαχειριστούμε σωστά όλες τις απαιτούμενες καταστάσεις.

Επίλυση των προβλημάτων

Για την επίλυση των προβλημάτων που παρουσιάστηκαν παραπάνω ακολουθήσαμε διάφορες κατευθύνεις και δοκιμάσαμε διάφορες υλοποιήσεις. Για άλλη μια φορά, σε κάθε προσπάθεια διόρθωσης των προβλημάτων προέκυπταν νέα προβλήματα ή διογκώνονταν τα ήδη υπάρχοντα.

Ύστερα από αρκετές αποτυχημένες προσπάθειες για τη διόρθωση των επιμέρους κομματιών και καλύτερη οργάνωση του κώδικα, αντιληφθήκαμε ότι θα ήταν προτιμότερο να εγκαταλείψουμε την ιδέα του παιχνιδιού F.E.A.R αφού οι διευκολύνσεις που παρείχε η γλώσσα προγραμματισμού C++ δεν τις παρείχε η C#. Έπρεπε λοιπόν να βρούμε μια εναλλακτική υλοποίηση των γεγονότων που θα κάλυπτε τους στόχους που είχαμε θέσει και θα μας επέτρεπε να χρησιμοποιήσουμε τα χαρακτηριστικά της γλώσσας C# για την καλύτερη εξυπηρέτηση των αναγκών μας.

Αποφασίσαμε λοιπόν κάθε γεγονός να το υλοποιήσουμε ως κλάση στην οποία να αποθηκεύονται το όνομα του γεγονότος καθώς και τα απαραίτητα αντικείμενα που χρειάζονταν για αυτό. Εγκαταλείψαμε την ιδέα των διαφορετικών τύπων και αποφασίσαμε να χρησιμοποιήσουμε τα γενικά αντικείμενα που απαρτίζουν τις οντότητες στη Unity3D (αντικείμενα τύπου GameObject). Αυτό μας διευκόλυνε αρκετά και βοήθησε στην καλύτερη οργάνωση του κώδικα, αφού κάθε GameObject παρέχει όλες τις απαραίτητες πληροφορίες που το αφορούν.

Μελετώντας περισσότερο τη γλώσσα C# μάθαμε πως εκτελεί διαχείριση αντικειμένων με αναφορές (reference), γεγονός το οποίο καθησύχασε προσωρινά την ανησυχία μας για μεγάλες απαιτήσεις μνήμης.

4.3.1.2 Υλοποίηση του iThinkFact ως κλάση και ύπαρξη παράγωγων κλάσεων.

Προσπάθεια

Σε αυτήν τη φάση της υλοποίησης κάθε fact αποτελούσε μία διαφορετική κλάση και όλες παράγονταν από τη βασική κλάση iThinkFact. Η iThinkFact παρείχε βασικές συναρτήσεις σύγκρισης, πρόσβασης και εγγραφής.

Κάθε κλάση που αντιπροσώπευε ένα γεγονός κληρονομούσε από τη βασική κλάση αυτές τις εικονικές συναρτήσεις και είτε τις χρησιμοποιούσε αυτούσιες ή υλοποιούσε τις δικές της με τη χρήση υπερφόρτωσης (override method).

Σκέψεις και εκτιμώμενα πλεονεκτήματα

Με την υλοποίηση κάθε γεγονότος ως διαφορετική κλάση καταφέραμε να οργανώσουμε ικανοποιητικά και αποτελεσματικά τον κώδικα και να έχουμε εύκολη διαχείριση των απαιτούμενων πληροφοριών, εξυπηρετώντας έτσι δύο από τους βασικούς στόχους που είχαμε θέσει από την αρχή της πτυχιακής μας.

Με την οργάνωση του κώδικα υπό μορφή κλάσεων ξεπεράσαμε τα σχεδιαστικά προβλήματα καθώς και τα προβλήματα υλοποίησης που είχαν εμφανιστεί με την προηγούμενη μορφή του κώδικα, αφού απλοποιήσαμε τη διαδικασία διατήρησης των απαιτούμενων πληροφοριών και απλοποιήθηκαν και οι περισσότερες συναρτήσεις που τις χρησιμοποιούσαν.

Ταυτόχρονα με την εγκατάλειψη της αρχικής προσέγγισης καταφέραμε να απαλλαγούμε από αρκετά κομμάτια κώδικα που έκαναν δυσνόητη την τεκμηρίωση και τον κώδικα, κρατώντας όσο το δυνατόν καθαρότερη τη μορφή του καινούριου κώδικα διευκολύνοντας την επαναχρησιμοποίηση και τη μελλοντική ανάπτυξή του .

Προβλήματα και δυσκολίες στην υλοποίηση

Επειδή τα γεγονότα είναι διαφορετικά μεταξύ τους και περιλαμβάνουν το καθένα ξεχωριστό αριθμό ορισμάτων, έπρεπε στη βασική κλάση να εισάγουμε πολλές διαφορετικές συναρτήσεις πρόσβασης (virtual accessors) για να εξυπηρετηθούν όλες οι παράγωγες κλάσεις και να έχουμε πρόσβαση στα διαφορετικά GameObjects.

Επέκταση αυτού του προβλήματος ήταν ότι αντιμετωπίσαμε δυσκολία στην υλοποίηση των συγκρίσεων αφού έπρεπε σε κάθε περίπτωση να προβλέψουμε το διαφορετικό αριθμό ορισμάτων για να αποφύγουμε πιθανά σφάλματα στην εκτέλεση.

Επίλυση των προβλημάτων

Για να ξεπεράσουμε ο πρόβλημα που είχε δημιουργηθεί χρησιμοποιήσαμε την αφαιρετική μέθοδο και απλοποιήσαμε τη συνάρτηση πρόσβασης στη βασική κλάση iThinkFact. Έτσι, φτιάξαμε μια συνάρτηση πρόσβασης που δεχόταν ως όρισμα έναν αριθμό και επέστρεφε το αντίστοιχο αντικείμενο από τη συλλογή των GameObjects.

4.3.1.3 Υλοποίηση του iThinkFact ως μοναδικής κλάσης.

Η απλοποίηση της συνάρτησης πρόσβασης είχε ως αποτέλεσμα να ακολουθήσουν μία σειρά από άλλες απλοποιήσεις που έφερε τον κώδικα του iThinkFact στην τελική του μορφή.

Βασιζόμενοι στη συνάρτηση πρόσβασης αλλάξαμε τη συνάρτηση δημιουργίας της κλάσης iThinkFact και χρησιμοποιήσαμε συνάρτηση με μεταβλητό αριθμό ορισμάτων (variadic function). Ακόμα αλλάξαμε τον τρόπο αποθήκευσης των δεδομένων και τα μετατρέψαμε σε λίστες από GameObjects. Η καινούρια συνάρτηση δέχεται ως όρισμα το όνομα του γεγονότος ως συμβολοσειρά, μια προαιρετική λογική μεταβλητή που καθορίζει αν το λεκτικό είναι θετικό ή αρνητικό (αν δεν οριστεί, το λεκτικό θεωρείται θετικό) και μια συλλογή αντικειμένων που αντιστοιχούν σε στιγμιότυπα του κόσμου.

Αυτή η αλλαγή οδήγησε στην απλοποίηση των υπόλοιπων συναρτήσεων της κλάσης iThinkFact καθώς και στην κατάργηση των παράγωγων κλάσεων της iThinkFact. Κάθε fact μπορεί να περιγραφεί από το όνομά του το οποίο το καθορίζει. Ταυτόχρονα αποθηκεύονται στη λίστα όλα τα απαιτούμενα GameObjects τα οποία είναι άμεσα διαχειρίσιμα, άρα δεν υπήρχε πλέον η ανάγκη για τη διατήρηση των διαφορετικών κλάσεων. Επίσης απλοποιήσαμε στο ελάχιστο δυνατό την παρουσίαση γνώσης κάνοντας απλή στην κατανόηση και στη διαχείριση.

Συνεπώς, με τη συνολική απλοποίηση της κλάσης iThinkFact καταφέραμε να κάνουμε τον κώδικα ευανάγνωστο, εύχρηστο και άμεσα αξιοποιήσιμο και επαναχρησιμοποιήσιμο. Ταυτόχρονα, μπορεί εύκολα να μελετηθεί και να επεκταθεί από οποιονδήποτε έχει προγραμματιστικές γνώσεις και θέλει να οπτικοποιήσει ή να χρησιμοποιήσει το σχεδιασμό με STRIPS.

4.3.2 Προβλήματα στην υλοποίηση του iThinkAction

4.3.2.1 ThinkAction με precondition και effects ως iThinkState

Προσπάθεια

Κατά τη δημιουργία των ενεργειών προσπαθήσαμε να δημιουργήσουμε κάθε ενέργεια ώστε να περιέχει προϋποθέσεις και επιδράσεις τα οποία να είναι το καθένα ένα iThinkState.

Έχοντας κατανοήσει τη λειτουργία και τη χρησιμότητα τους, προσπαθήσαμε να ομαδοποιήσουμε τα γεγονότα που περιγράφουν τις ενέργειες.

Σκέψεις και εκτιμώμενα πλεονεκτήματα

Θεωρούσαμε πως οργανώνοντας τις προϋποθέσεις και τις επιδράσεις ως καταστάσεις θα είχαμε αρκετές ευκολίες στην υλοποίηση και θα μπορούσαμε να διαχειριστούμε καλύτερα τη συνολική πληροφορία. Έχοντας ήδη έτοιμες τις συναρτήσεις διαχείρισης των καταστάσεων εκτιμούσαμε πως θα μπορούσαμε να εισάγουμε και να διαγράψουμε εύκολα πληροφορίες από τις προϋποθέσεις και τις επιδράσεις από την κατάσταση που εξέταζε ο αλγόριθμός μας χωρίς να χρειαστούμε επιπλέον συναρτήσεις διαχείρισης, καθιστώντας απλούστερη την εκτέλεση της κατάστρωσης σχεδίου.

Θεωρώντας δυστυχώς λανθασμένα ότι η συνολική γνώση που έχει ο πράκτορας είναι ανεξάρτητη από την παραγωγή των διαφόρων ενεργειών, εκτιμούσαμε ότι κατά την εκτέλεση του προγράμματος ο πράκτορας θα έβρισκε αυτόματα τα απαραίτητα αντικείμενα και τις αναγκαίες πληροφορίες για να συμπληρώσει και να συνδέσει τα κενά γνώσης που θα είχαν αυτές.

Προβλήματα και δυσκολίες στην υλοποίηση

Στη υλοποίηση των ενεργειών αντιμετωπίσαμε αρκετά προβλήματα και δυσκολίες οι οποίες μας καθυστέρησαν αλλά και μας βοήθησαν να κατανοήσουμε σε βάθος τις έννοιες των προϋποθέσεων και των επιδράσεων του STRIPS.

Μια δυσκολία που προκύπτει κατά την υλοποίηση των προϋποθέσεων και των επιδράσεων είναι πως χρησιμοποιούνται αποκλειστικά στο σχεδιασμό και δεν αποτελούν κομμάτι της συνολικής διαδικασίας. Όταν ελέγχεται μία ενέργεια, ελέγχονται οι προϋποθέσεις της για να καθοριστεί αν είναι εφαρμόσιμη. Αν μπορεί να εφαρμοστεί, οι επιδράσεις αξιοποιούνται και παράγουν, σε συνδυασμό με τη τωρινή κατάσταση που

εξετάζεται, την νέα κατάσταση που εισάγεται στο σύνορο αναζήτησης. Ουσιαστικά, καμία από τις εκτιμώμενες ενέργειες δεν ενεργοποιείται πραγματικά.

Μια ακόμα επιλογή που μας δυσκόλεψε ήταν πως αρχικά θεωρήσαμε τη γνώση που έχει ο πράκτορας ανεξάρτητη από τη δημιουργία των ενεργειών και δεν τη λαμβάναμε υπόψη μας. Αντιθέτως, είναι απαραίτητη για τη δημιουργία αυτών αφού καθορίζει το σύνολο των ενεργειών που μπορεί να εκτελέσει ο πράκτορας. Τα αντικείμενα που γνωρίζει ο πράκτορας καθορίζουν και τις διαθέσιμες ενέργειες που μπορεί να πραγματοποιήσει για να λειτουργεί στο παιχνίδι, άρα είναι απαραίτητη η αξιοποίηση της διαθέσιμης γνώσης για τη δημιουργία τους. Αποτέλεσμα του προβλήματος αυτού ήταν ότι αδυνατούσαμε να παράγουμε το απαραίτητο συνδυασμό αντικειμένων για τη δημιουργία των preconditions και των effects αφού δεν είχαμε προβλέψει να έχουμε πρόσβαση στη γνώση του πράκτορα.

Μια παράλειψη στο σχεδιασμό μας παρουσιάστηκε στη σύγκριση που απαιτείται μεταξύ των προϋποθέσεων και των γεγονότων της τρέχουσας κατάστασης για να αποφασιστεί αν μία ενέργεια είναι εφαρμόσιμη. Ο τρόπος που εκτελούσαμε τις συγκρίσεις για τις καταστάσεις απαιτούσε οι προϋποθέσεις μίας κατάστασης να είναι ίσες σε αριθμό και ίδιες με τα γεγονότα σε αυτήν για να προκύψει αληθής η σύγκριση. Στο STRIPS, τη λογική την οποία ακολουθούσαμε, οι προϋποθέσεις πρέπει να αποτελούν μόνο ένα υποσύνολο των γεγονότων της κατάστασης για να είναι εφαρμόσιμη η ενέργεια.

Στις ενέργειες, σε συνδυασμό με τη μορφή που οργανώναμε τα γεγονότα μέχρι εκείνη τη στιγμή, ήταν αδύνατο να συνδέσουμε τις οντότητες του περιβάλλοντος με τα αντικείμενα που περιγράφονταν στο σχεδιαστή, με αποτέλεσμα να εντείνεται το πρόβλημα μη αξιοποίησης και παρουσίασης αναγκαίας γνώσης για την ορθή λειτουργία της κατάστρωσης σχεδίου.

Κατά την εκτέλεση του σχεδιασμού έγιναν αντιληπτά και άλλα προβλήματα που προέκυπταν από την επιλογή να αναπαραστήσουμε τις προϋποθέσεις και τις επιδράσεις ως αντικείμενα τύπου iThinkState. Το σοβαρότερο πρόβλημα προέκυψε με τις επαναλήψεις που έπρεπε να κάνει ο σχεδιαστής.

Σε κάθε επανάληψη έπρεπε να δημιουργούνται καινούριες προϋποθέσεις και επιδράσεις και να αποθηκεύονταν στην ενέργεια για να χρησιμοποιηθούν σε επόμενη επανάληψη. Το πρόβλημα εδώ ήταν ότι αφορούσαν διαφορετικά GameObjects, τα οποία με την υλοποίηση που είχαμε ακολουθήσει μέχρι εκείνη τη στιγμή ήταν πολλές φορές αδύνατο να τα αποθηκεύσουμε σε κάθε κατάλληλη ενέργεια. Επίσης, όταν άλλαζε μία ενέργεια η προηγούμενη χανόταν αφού αποθηκεύαμε μόνο μία τη φορά.

Αυτό είχε ως αποτέλεσμα να μην έχουμε διαθέσιμο το σύνολο των ενεργειών που χρειαζόμασταν για να γίνει σωστά ο σχεδιασμός.

Η δημιουργία των ενεργειών, ανεξαρτήτως της συνολικής γνώσης που έχει ο πράκτορας, δημιούργησε κενά στις προϋποθέσεις και τις επιδράσεις των καινούριων ενεργειών που προέκυπταν, τα οποία δεν μπορούσαν να καλυφθούν. Εμφανίστηκε επιπλέον το πρόβλημα της παρουσίασης αρνητικής γνώσης στις επιδράσεις και στις προϋποθέσεις και το πως αυτή θα μπορούσε να είναι άμεσα διαχειρίσιμη από το πρόγραμμα. Η αρνητική γνώση στο STRIPS χρησιμοποιείται για να αφαιρεθούν γεγονότα από την κατάσταση που ελέγχει ο σχεδιαστής και να δημιουργηθούν νέες καταστάσεις για να συνεχιστεί ο σχεδιασμός. Εμείς μέχρι εκείνη τη στιγμή δεν είχαμε προβλέψει την αρνητική γνώση και τον τρόπο παρουσίασης της στο πρόγραμμά μας.

Οι προσπάθειες επίλυσης των παραπάνω προβλημάτων είχαν ως αποτέλεσμα αρκετά σχεδιαστικά αδιέξοδα και προβλήματα υλοποίησης τα οποία, σε συνδυασμό με την προσπάθεια αντιμετώπισής τους, οδήγησαν σε δημιουργία δυσνόητου κώδικα και δύσκολου στην τεκμηρίωση. Αποφασίσαμε λοιπόν να επιλύσουμε μεθοδικά τα προβλήματα που προέκυπταν και να κάνουμε απλοποιήσεις όπου χρειαζόταν για να μην παρεκκλίνουμε από τους στόχους που είχαμε θέσει.

Επίλυση των προβλημάτων

Σε αυτήν τη φάση της υλοποίησης λύσαμε τα περισσότερα προβλήματα που αντιμετωπίζαμε και προσπαθήσαμε να οργανώσουμε καλύτερα τον κώδικα για να είναι πιο κατανοητός και εύχρηστος.

Για να αντιμετωπίσουμε το πρόβλημα που είχαμε με τις συγκρίσεις στις διάφορες καταστάσεις, απλοποιήσαμε τις προϋποθέσεις και τις επιδράσεις στην κλάση iThinkAction και τις αναγάγαμε σε απλές λίστες από iThinkFact. Έχοντας κατανοήσει τη σημασία τους στη κατάστρωση σχεδίου και το βασικό πρόβλημα που είχαμε στην αναπαράσταση της αρνητικής γνώσης, αποφασίσαμε να δημιουργήσουμε τέσσερις διαφορετικές λίστες για την παρουσίαση της θετικής και της αρνητικής γνώσης τόσο στις προϋποθέσεις όσο και στις επιδράσεις.

Αποφασίσαμε επίσης να απλοποιήσουμε την κλάση iThinkFact προκειμένου να διαχειρίζεται πλέον GameObjects και να εγκαταλείψουμε την πολύπλοκη δομή που περιγράφηκε στο εδάφιο 4.3.1.1. Με αυτόν τον τρόπο απλοποιήσαμε και τις συγκρίσεις μεταξύ των facts και των states. Ακόμα υλοποιήσαμε τη συνάρτηση ελέγχου

εφαρμοσιμότητας, η οποία πραγματοποιεί τη σωστή σύγκριση μεταξύ της τρέχουσας κατάστασης και των προϋποθέσεων.

Με την απλοποίηση του iThinkFact επιτύχαμε να συνδέσουμε τα αντικείμενα του παιχνιδιού απευθείας με τα δεδομένα και τα στοιχεία του προγράμματος, καταφέρνοντας έτσι να ξεπεράσουμε το βασικό πρόβλημα που αντιμετωπίζαμε από την αρχή της υλοποίησης του σχεδιαστή μας.

4.3.2.2 iThinkAction με precondition και effects ως τέσσερις λίστες.

Προσπάθεια

Για την ορθότερη παρουσίαση της αρνητικής γνώσης αποφασίσαμε να δημιουργήσουμε τέσσερις λίστες για τις προϋποθέσεις και τις επιδράσεις κάθε ενέργειας. Δύο λίστες για να παρουσιάζεται η θετική και η αρνητική γνώση στις προϋποθέσεις και αντίστοιχα δύο λίστες για τις επιδράσεις.

Σκέψεις και εκτιμώμενα πλεονεκτήματα

Οι τέσσερις λίστες δημιουργούσαν μία ομαδοποίηση δεδομένων που τη δεδομένη χρονική στιγμή θεωρήθηκε καλή. Η επιλογή μας αυτή απλοποίησε την επεξεργασία των ενεργειών, αφού κάθε επιμέρους κομμάτι που αφορούσε την ενέργεια ήταν ξεχωριστό.

Σκοπός μας ήταν σε κάθε επανάληψη της κατάστρωσης σχεδίου να ελέγχουμε πρώτα τις δύο λίστες με τις προϋποθέσεις και μετά τις δύο λίστες για τις επιδράσεις, εφόσον η συνάρτηση ελέγχου εφαρμοσιμότητας είχε επιστρέψει θετικό αποτέλεσμα. Στην εφαρμογή των επιδράσεων θεωρούσαμε πως ήταν πολύ πιο εύκολο να διατρέξει δύο διαφορετικές λίστες για την εισαγωγή και να εξαγωγή διαφορετικών γεγονότων από μία κατάσταση και φαινόταν τη δεδομένη στιγμή η καλύτερη λύση.

Προβλήματα και δυσκολίες στην υλοποίηση

Τα προβλήματα που παρουσιάστηκαν σε αυτή τη φάση της υλοποίησης ήταν κυρίως θέματα υλοποίησης και κατανόησης.

Χρειάστηκε λίγη παραπάνω μελέτη και συζήτηση σχετικά με το πώς παρουσιάζει το STRIPS την αρνητική γνώση. Η αρνητική γνώση εμφανίζεται μόνο στις επιδράσεις κάθε ενέργειας, ενώ η γνώση που δεν έχει ο αλγόριθμος δε χρειάζεται να παρουσιαστεί στις προϋποθέσεις. Όλη η γνώση που απαιτείται για τις προϋποθέσεις παρουσιάζεται μόνο

με θετικά λεκτικά, αφού μας ενδιαφέρει μόνο η γνώση που χρειάζεται να έχει ο αλγόριθμος για να πραγματοποιήσει μία ενέργεια.

Από την άλλη πλευρά η αρνητική γνώση πρέπει να εμφανίζεται στις επιδράσεις των διαφορετικών ενεργειών αφού μία ενέργεια μπορεί να έχει και αρνητική επίδραση στον πράκτορα, υπονοώντας τη διαγραφή της σχετικής γνώσης. Για παράδειγμα, αν ο πράκτορας γνωρίζει το γεγονός isHolding(box) και ελέγχεται η ενέργεια dropBox(box) με προϋποθέσεις isHolding(box) και επιδράσεις ~isHolding(box)^emptyHands, τότε η επόμενη κατάσταση που θα δημιουργηθεί από την εφαρμογή της ενέργειας περιλαμβάνει μόνο τη γνώση emptyHands. Η αρνητική γνώση στις επιδράσεις είναι καθοριστική για την επόμενη κατάσταση που θα δημιουργηθεί από την εφαρμογή της ενέργειας και συμβάλει στην ολοκλήρωση του σχεδιασμού.

Λαμβάνοντας υπόψη μας τα παραπάνω παρατηρήσαμε ότι η εισαγωγή αρνητικής γνώσης στις προϋποθέσεις θα δημιουργούσε αρκετά μεγάλο φόρτο κατά την εκτέλεση του προγράμματος και θα καθυστερούσε τη διαδικασία του σχεδιασμού, αφού θα απέκλινε από το απλοποιημένο μοντέλο που προτείνει το STRIPS.

Επιπλέον, δεν είχαμε ακόμα συνδέσει στη γνώση που έχει ο πράκτορας με τη δημιουργία των ενεργειών με αποτέλεσμα να έχουμε, παρόμοια με την προηγούμενη φάση υλοποίησης, προβλήματα με ελλιπή γνώση την οποία δεν μπορούσαμε να συμπληρώσουμε για να ολοκληρωθεί επιτυχώς η διαδικασία.

Επίλυση των προβλημάτων

Για την επίλυση του προβλήματος της παρουσίασης αρνητικής γνώσης αποφασίσαμε να απλοποιήσουμε περαιτέρω τον υπάρχοντα κώδικα. Διατηρήσαμε μόνο δύο λίστες για την παρουσίαση της γνώσης, μία που θα αποτελούσε τις προϋποθέσεις και μία που θα αποτελούσε τις επιδράσεις. Αποφασίσαμε να μην εμφανίζουμε αρνητική γνώση στις προϋποθέσεις (ακολουθώντας περισσότερο τη λογική του STRIPS), ενώ στις επιδράσεις θα παρουσιαζόταν αρνητική και θετική γνώση που θα καθοριζόταν βάσει μιας μεταβλητής στη κλάση iThinkFact. Τη λογική μη εμφάνισης αρνητικής γνώσης την εφαρμόσαμε και στην κλάση iThinkState απλοποιώντας αρκετά τον κώδικά της.

Για να δηλώσουμε την αρνητική γνώση στα γεγονότα εισάγαμε τη λογική μεταβλητή ‘positive’ στη κλάση iThinkFact. Η προεπιλεγμένη τιμή της είναι ‘αληθής’ και ο χρήστης

θα την ορίσει ‘ψευδής’ στη περίπτωση που θέλει να δηλώσει πως το γεγονός πρέπει να αφαιρεθεί από την αντίστοιχη κατάσταση μετά την εφαρμογή της ενέργειας.

Συνδέσαμε επομένως τη δημιουργία των ενεργειών με τη γνώση του πράκτορα, λύνοντας ένα από τα μεγαλύτερα προβλήματα που αντιμετωπίσαμε κατά την εκτέλεση του σχεδιασμού. Ουσιαστικά, αλλάξαμε τη σειρά με την οποία δημιουργούνται οι ενέργειες. Στην αρχή είχαμε πρώτα τη δημιουργία των ενεργειών και μετά την απόκτηση γνώσης των αντικειμένων του παιχνιδιού από τον πράκτορα, ενώ πλέον αυτός αντλεί αρχικά γνώση από το περιβάλλον του παιχνιδιού (μέσω για παράδειγμα της κλάσης iThinkSensorySystem) και τότε παράγει τις διαθέσιμες ενέργειες για την εφαρμογή του σχεδιασμού, με μια διαδικασία που θα αναλυθεί στη περιγραφή υλοποίησης της κλάσης iThinkActionSet.

4.3.2.3 Ολοκλήρωση του iThinkAction

Κάθε action στη βιβλιοθήκη iThink περιγράφεται πλέον από τα preconditions και τα effects του, και προαιρετικά με ένα όνομα. Κάθε νέα δυνατή action θα κληρονομεί από την iThinkAction τέσσερεις συναρτήσεις τις οποίες θα αναλύσουμε παρακάτω: initPreconditions(), initEffects(), applyEffects() και isApplicable().

Στις συναρτήσεις δημιουργίας χρησιμοποιούμε την ιδιότητα των μεταβλητών ορισμάτων για να περάσουμε στη δημιουργία της ενέργειας το απαραίτητο πλήθος πληροφοριών είναι απαραίτητη για τις προϋποθέσεις και τις επιδράσεις. Εκεί καλούμε και τις δύο βασικές συναρτήσεις που χρησιμεύουν για την υλοποίηση αυτών. Οι συναρτήσεις που τις δημιουργούν (initPreconditions() και initEffects() αντίστοιχα) αρχικοποιούν τις αντίστοιχες λίστες καλώντας τις συναρτήσεις της βασικής κλάσης και μετά εισάγουν σε αυτές τα αντίστοιχα γεγονότα. Επειδή δεν υπάρχει αυτοματοποιημένος τρόπος για τη δημιουργία των προϋποθέσεων και των επιδράσεων, ο χρήστης πρέπει μονάχα να δηλώσει τα γεγονότα που περιγράφουν τις αντίστοιχες λίστες.

Η συνάρτηση isApplicable() ελέγχει αν οι προϋποθέσεις μιας ενέργειας είναι υποσύνολο της τρέχουσας κατάστασης που εξετάζεται. Αν ο έλεγχος αυτός είναι επιτυχής, σημαίνει πως η συγκεκριμένη ενέργεια είναι εφαρμόσιμη.

Η συνάρτηση applyEffects() εξασφαλίζει την εφαρμογή των επιδράσεων της κάθε ενέργειας στην τρέχουσα κατάσταση προκειμένου να παράγει μία επόμενη κατάσταση που θα χρησιμοποιηθεί στη συνέχεια του σχεδιασμού για την εύρεση του τελικού στόχου.

4.3.3 Προβλήματα στην υλοποίηση του iThinkActionSet

4.3.3.1 Το iThinkActionSet με τη χρήση Reflection

Προσπάθεια

Η αρχική προσπάθεια για ένα οργανωμένο σύνολο από το οποίο ο πράκτορας θα αντλεί τις ενέργειες και θα τις ελέγχει για την υλοποίηση του πλάνου, προσεγγίστηκε με την αρχική μας ιδέα ότι οι ενέργειες είναι ανεξάρτητες από τη γνώση που έχει. Η ιδέα αυτή μας οδήγησε στη χρήση του Reflection για την εύρεση και τη δημιουργία όλων των ενεργειών που μπορεί να εφαρμόσει ο πράκτορας. Χρησιμοποιώντας Reflection προσπαθούσαμε να αντλήσουμε κατά την εκτέλεση του προγράμματος όλα τα αντικείμενα τύπου iThinkAction που ο χρήστης είχε δηλώσει, να τα δημιουργήσουμε ένα προς ένα και να τα εισάγουμε σε μία λίστα από την οποία ο πράκτορας θα αξιοποιεί κάθε φορά αυτά που χρειάζεται.

Εκτιμώμενα πλεονεκτήματα

Με αυτόν τον τρόπο προσπαθούσαμε να οργανώσουμε και να δημιουργούμε τα actions με τον τρόπο που τα δημιουργεί και τα οργανώνει το F.E.A.R. Η μεγάλη διαφορά είναι πως το F.E.A.R χρησιμοποιεί τον προεπεξεργαστή της C++ και έτσι δημιουργεί κλάσεις, συναρτήσεις και αντικείμενα (μέσω μοτίβων κατασκευής) των ενεργειών τις οποίες μπορεί να διαχειριστεί από τη φάση της μεταγλώττισης κιόλας, ενώ ο χρήστης χρειάζεται να ορίσει μόνο τα ονόματα των ενεργειών υπό μορφή συμβολοσειρών στις αντίστοιχες μακροεντολές. Εμείς θεωρούσαμε πως, αφού δεν έχουμε στη διάθεσή μας προεπεξεργαστή, θα μπορούσαμε να υλοποιήσουμε την ίδια διαδικασία με τη χρήση του Reflection, απλοποιώντας τη παραγωγή του κάθε ενέργειας στη χρήση μίας μόνο συμβολοσειράς που θα αξιοποιούσε το πρόγραμμα μας, απλοποιώντας κατά πολύ τη χρήση του.

Προβλήματα και δυσκολίες στην υλοποίηση

Το πρόβλημα με τη χρήση του Reflection δεν έγκειται στη δυσκολία που είχε η υλοποίηση του κατάλληλου κώδικα αλλά στο γεγονός ότι μέχρι εκείνη τη στιγμή δεν είχαμε συνδέσει τη γνώση που έχει ο πράκτορας με τη δημιουργία των ενεργειών, πράγμα που προκαλούσε όλα τα προαναφερθέντα προβλήματα τα οποία παρουσιάζονταν και σε αυτή τη φάση. Αρκετά αρνητικό ήταν και το γεγονός πως το Reflection ως τεχνική πρόσθετε αρκετά μεγάλο φόρτο στο πρόγραμμα και η δημιουργία

των ενεργειών θα ήταν ιδιαίτερα χρονοβόρα, πράγμα απαγορευτικό για τη χρήση του σχεδιαστή σε περιβάλλον βιντεοπαιχνιδιών.

Επίλυση των προβλημάτων

Όλη η δομή του iThinkActionSet άλλαξε και προσαρμόστηκε στο νέο κώδικα που υλοποιήσαμε για την ολοκλήρωση της κατάστρωσης σχεδίου. Η σύνδεση τη γνώσης του πράκτορα με τη δημιουργία των ενεργειών δημιούργησε επιπλέον την ανάγκη να αυτοματοποιήσουμε τη δημιουργία τους με έναν τρόπο προσιτό στο χρήστη.

4.3.3.2 Κατάργηση iThinkActionSet και δημιουργία iThinkActionSchemas

Έχοντας πλέον ως άξονα αναφοράς την ανάγκη να κάνουμε τη δημιουργία των ενεργειών πιο προσιτή και εύκολη για το χρήστη προσπαθήσαμε να βρούμε ένα τρόπο να αξιοποιούμε αποτελεσματικά την υπάρχουσα γνώση του πράκτορα, να τη μορφοποιούμε κατάλληλα και να τη χρησιμοποιήσουμε στην παραγωγή ολοκληρωμένων, από πλευράς γνώσης, ενεργειών που θα μπορούν να χρησιμοποιηθούν άμεσα από το σχεδιαστή.

Η αρχική μας προσέγγιση ήταν από την πλευρά του κώδικα. Θέλαμε να παρέχουμε στο πρόγραμμα τη δυνατότητα, κατά τη δημιουργία των ενεργειών, να έχει συγκεντρωμένη όλη τη γνώση για να τη διαχειριστεί άμεσα περιορίζοντας το ρόλο και την επίδραση του χρήστη σε αυτή. Αυτό μας οδήγησε στην κατάργηση του iThinkActionSet και στη δημιουργία δύο νέων κλάσεων, της iThinkActionSchemas και της iThinkActionManager.

Η iThinkActionManager έχει ως στόχο να δημιουργεί και να παρέχει λίστα διαθέσιμων ενεργειών στον πράκτορα. Για τη δημιουργία της λίστας αυτής αξιοποιεί τις συναρτήσεις του iThinkActionSchemas.

Η iThinkActionSchemas είναι η κλάση που ομαδοποιεί τη γνώση του πράκτορα και χρησιμοποιείται για τη δημιουργία όλων των κατάλληλων ενεργειών που είναι διαθέσιμες στον πράκτορα με τη διαθέσιμη γνώση την ώρα της δημιουργίας. Θεωρήσαμε πως θα ήταν αρκετά κατανοητό αν ο χρήστης εισήγαγε ένα πίνακα συμβολοσειρών που θα αντιπροσώπευε το σύνολο των διαθέσιμων ενεργειών. Κάθε συμβολοσειρά έχει τη δομή:

<όνομα action>-<αριθμός ορισμάτων constructor>(-tag~<όνομα tag>)*

Σχήμα 4. Πρότυπο περιγραφής ενεργειών (action schema)

Το πεδίο <όνομα action> είναι το όνομα της κλάσης που υλοποιεί την ενέργεια, ενώ το πεδίο <αριθμός ορισμάτων constructor> είναι το σύνολο των ορισμάτων που δέχεται η συνάρτηση δημιουργίας της. Η λίστα από πεδία tag~<tag αντικειμένου> αντιπροσωπεύουν την απαραίτητη γνώση που χρειάζονται οι ενέργειες για να υλοποιηθούν. Η κατηγοριοποίηση των οντοτήτων του κόσμου με μια ετικέτα (που στο περιβάλλον Unity3D περιγράφεται ως Tag) απλοποιεί πάρα πολύ τη διαδικασία κατάστρωσης σχεδίου και έχει δυνητικά μεγάλη χρησιμότητα και στο προγραμματισμό που υπόλοιπου παιχνιδιού. Για παράδειγμα, το avatar του παίκτη μπορεί να έχει ετικέτα “player” και οι οντότητες που αντιστοιχούν σε ανταγωνιστές να έχουν ετικέτα “enemy”.

Για παράδειγμα, στον κόσμο του BlocksWorld, ο πράκτορας έχει διαθέσιμη την ενέργεια ActionBwStack με την οποία μπορεί να στοιβάξει ένα μπλοκ πάνω σε ένα άλλο. Η περιγραφή αυτής της ενέργειας ως actionSchema είναι η εξής

```
ActionBwStack-2-Tag~block-Tag~block
```

και υποδηλώνει πως το όνομα της κλάσης είναι ActionBwStack και έχει δύο παραμέτρους, τα οποία πρέπει να είναι GameObjects με ετικέτα “block”.

Κάθε συμβολοσειρά εισάγεται τελικά μέσα στο σύστημα και περνάει από μία διαδικασία επεξεργασίας και διαμόρφωσης (parsing) προκειμένου να καθοριστούν οι απαραίτητες πληροφορίες που χρειάζεται η τρέχουσα ενέργεια. Πλέον, το σύστημα έχει ολοκληρώσει τη συλλογή απαραίτητων πληροφοριών για τη δημιουργία των διαθέσιμων ενεργειών.

Μια μεγάλη διαφορά με τις γλώσσες Λογικού Προγραμματισμού που εφαρμόζονται κατά κανόνα στην ακαδημαϊκή κοινότητα είναι η ιδιότητα της ενοποίησης (unification) που παρέχεται στα αντίστοιχα συστήματα. Προφανώς, αυτή η λειτουργία πρέπει να αντικατασταθεί χειρονακτικά στις υλοποιήσεις άλλων γλωσσών ή να αποφευχθεί εξολοκλήρου, ανάλογα με την υλοποίηση του συστήματος. Στη δημιουργία του iThink αναλάβαμε να φτιάξουμε ένα σύστημα το οποίο θα ακολουθεί τις αρχές των σχεδιαστών ακαδημαϊκής φύσης προσαρμοσμένες στις ανάγκες μιας σύγχρονης μηχανής παιχνιδιών. Συνεπώς η απλοποίηση του συστήματος απαιτούσε τη δημιουργία της συνάρτησης Unifier(), η οποία αναλαμβάνει την ενοποίηση των προαναφερθέντων δεδομένων.

Το τελικό αποτέλεσμα είναι πως ο χρήστης αναλαμβάνει μονάχα να επεξεργαστεί τη συνάρτηση iThinkActionSchemas::actionGenerator() και, με βάση το όνομα της συνάρτησης, να δημιουργήσει τη σχετική συνάρτηση ανάλογα με το πλήθος των παραμέτρων της. Για παράδειγμα, στον κόσμο του BlocksWorld έχουμε ορίσει τη δράση ActionBwStack, η οποία αναλαμβάνει να στοιβάξει ένα αντικείμενο πάνω σε ένα άλλο. Ο χρήστης αναλαμβάνει να εισάγει στη σχετική δομή επιλογής της actionGenerator() το εξής:

```
case "ActionBwStack":
    action = new ActionBwStack( "Stack", matrix[0], matrix[1] );
    tempActionList.Add( action );
    break;
```

Στην ενότητα των συμπερασμάτων θα συζητηθούν περαιτέρω βελτιώσεις και αυτοματοποιήσεις που μπορούν να υλοποιηθούν στη κλάση iThinkActionSchemas.

5. Επίλογος

5.1 Σύνοψη και συμπεράσματα

Όπως αναφέρθηκε βασικός στόχος της δημιουργίας της βιβλιοθήκης μας ήταν η μελέτη και υλοποίηση τεχνικών σχεδιασμού με ιδιαίτερη έμφαση στην εκπαιδευτική της χρησιμότητα. Μελετώντας σε βάθος τον κλασσικό σχεδιασμό και τη μέθοδο αναπαράστασης STRIPS υλοποιήσαμε τη βιβλιοθήκη iThink η οποία αποτελεί ένα πολύ καλό πρώτο στάδιο για την εισαγωγή ενός νέου χρήστη ή φοιτητή στον κόσμο του σχεδιασμού της τεχνητής νοημοσύνης.

Παρόλο που στην τωρινή της μορφή η βιβλιοθήκη απέχει αρκετά από έναν ολοκληρωμένο σχεδιαστή που χρησιμοποιείται σε σύγχρονες εφαρμογές, έχει τη δυνατότητα να λύσει οποιοδήποτε πρόβλημα σχεδιασμού. Φυσικά, για μεγάλα προβλήματα δεν είναι ιδιαίτερα αποδοτική. Αυτό συμβαίνει γιατί στην πλειοψηφία τους οι σύγχρονοι σχεδιαστές υλοποιούνται για πολύ συγκεκριμένες λειτουργίες με πλήθος βελτιστοποιήσεων, ενώ εμείς σχεδιάσαμε τη βιβλιοθήκη μας για να είναι γενικευμένη και σε θέση να λύσει οποιοδήποτε πρόβλημα σχεδιασμού.

Εκπληρώνοντας το δεύτερο στόχο που είχαμε θέσει, δηλαδή η βιβλιοθήκη να είναι όσο το δυνατόν πιο κατανοητή και απλή από άποψη κώδικα, παρέχουμε στον αναγνώστη μία ολοκληρωμένη και καλά οργανωμένη υλοποίηση με πλήθος σχολίων και καλή τεκμηρίωση την οποία μπορεί να μελετήσει σε βάθος. Μας δίνεται και εμάς πλέον η δυνατότητα να εμπλουτίσουμε την υπάρχουσα βιβλιοθήκη με πληθώρα επεκτάσεων, λειτουργιών και βελτιστοποιήσεων ώστε να τη μετατρέψουμε σε ένα πιο ολοκληρωμένο εργαλείο σχεδιασμού που θα μπορεί να αξιοποιηθεί για οποιαδήποτε εφαρμογή, διατηρώντας πάντα την απλότητα της.

Ελπίζουμε η βιβλιοθήκη iThink να χρησιμοποιηθεί από την ακαδημαϊκή κοινότητα για την πρακτική μελέτη προβλημάτων σχεδιασμού σε διαδραστικά περιβάλλοντα, ως εκπαιδευτικό εργαλείο για την κατανόηση του σχεδιασμού, ενώ ταυτόχρονα να αποτελέσει και ένα εργαλείο το οποίο να μπορεί να χρησιμοποιηθεί εύκολα από τους χρήστες για την υλοποίηση οποιασδήποτε ψυχαγωγικής ή εκπαιδευτικής εφαρμογής που απαιτεί σχεδιασμό.

5.2 Μελλοντικές επεκτάσεις

Παρουσιάζονται ορισμένες επεκτάσεις οι οποίες μελλοντικά μπορούν να ενταχθούν στη βιβλιοθήκη μας αλλά δεν ήταν μέρος των στόχων της παρούσας πτυχιακής:

- Υλοποίηση διαφορετικών αλγορίθμων σχεδιασμού, για να μπορεί η βιβλιοθήκη να χρησιμοποιηθεί και ως ένα εργαλείο για τη μελέτη κλιμάκωσης και σύγκρισης των αλγορίθμων σχεδιασμού για ένα πρόβλημα. Ιδιαίτερη έμφαση θα δώσουμε στην υλοποίηση του αλγορίθμου A* για τη βελτιστοποίηση του σχεδιασμού.
- Ολοκλήρωση του μεταφραστή για τα Action Schemas. Στη παρούσα φάση, η κλάση Action Schemas διαχειρίζεται μόνο ένα είδος πληροφορίας. Στόχος μας είναι να εμπλουτίσουμε την πληροφορία που θα μπορεί να διαχειριστεί δίνοντας τη δυνατότητα στο χρήστη να επιλέγει σε μία μορφή το χώρο αναζήτησης της κατάστρωσης σχεδίου για τη δημιουργία των ενεργειών.
- Καλύτερη οργάνωση στον τρόπο που πραγματοποιείται ο σχεδιασμός. Στην τωρινή μορφή η κατάστρωση σχεδίου πραγματοποιεί το πλάνο στην αρχή με το ξεκίνημα του αλγορίθμου. Στόχος μας είναι να εισάγουμε ένα χρονικό προϋπολογισμό για την κατάστρωση σχεδίου και να διασπάμε τη διαδικασία σχεδιασμού με βάση αυτόν τον προϋπολογισμό αξιοποιώντας ισομερώς του πόρους που παρέχει το Unity3D.
- Αξιοποίηση σύγχρονων τεχνικών διαμοιρασμού φόρτου εργασίας όπως η πολυνημάτωση (threading), για τη βελτιστοποίηση της εκτέλεσης του συνολικού προγράμματος και κατ' επέκταση της κατάστρωσης σχεδίου όπως και για καλύτερη αξιοποίηση των συνολικών πόρων και δυνατοτήτων του Unity3D.
- Πλήρης σύνδεση του iThink με το συντάκτη του Unity3D. Θέλοντας να διευκολύνουμε το χρήστη και να περιορίσουμε την επέμβασή του στον κώδικά έχουμε σκοπό να συνδέσουμε πλήρως τη βιβλιοθήκη iThink με το Unity3d και να δημιουργήσουμε κατάλληλες επιλογές (menu) και συναρτήσεις οι οποίες θα αυτοματοποιούν βασικές διαδικασίες τις οποίες τώρα καλείται να υλοποιήσει ο χρήστης και θα παρέχουν ευκολίες υλοποίησης.
- Δημιουργία κοινής διαμοιραζόμενης μνήμης (working memory) μεταξύ των διαφορετικών καταστρώσεων σχεδίου από διαφορετικούς πράκτορες. Όταν έχουμε πολλούς διαφορετικούς πράκτορες οι οποίοι δρουν στο ίδιο περιβάλλον υπάρχουν γεγονότα τα οποία είναι κοινά για όλους. Στην τωρινή μορφή της βιβλιοθήκης κάθε πράκτορας έχει τη δική του γνώση η οποία μπορεί να περιέχει κοινά στοιχεία με των υπολοίπων. Η επανάληψη τέτοιων στοιχείων δημιουργεί περιττό φόρτο στην εκτέλεση του προγράμματος. Στόχος μας είναι η δημιουργία μιας διαμοιραζόμενης μνήμης η οποία θα περιέχει την κοινή γνώση που έχουν οι πράκτορες και θα έχουν

πρόσβαση όλοι σε αυτή. Με αυτό τον τρόπο μειώνουμε την επαναλαμβανόμενη περιττή πληροφορία και το φόρτο εκτέλεσης.

- Με βάση το παραπάνω στοιχείο θα μπορέσουμε να εισάγουμε και πολλαπλούς πράκτορες για ένα πρόβλημα οι οποίοι θα δρουν γνωρίζοντας τις μεταβολές στο περιβάλλον τους. Στην τωρινή μορφή της βιβλιοθήκης κάθε πράκτορας δρα ανεξάρτητα από τους άλλους και δεν γνωρίζει τυχόν μεταβολές που έχουν γίνει στις γνώσεις του από τη δράση άλλων πρακτόρων.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Action	Ενέργεια
Action schema	Περιγραφή ενέργειας
After constraint	Μετά- περιορισμός
Automated Planning	Αυτοματοποιημένος/η Σχεδιασμός, Κατάστρωση σχεδίου
Backward search	Ανάστροφη/προς-τα-πίσω αναζήτηση
Before constraint	Πριν- περιορισμός
Best first search	Αναζήτηση πρώτα-στο-καλύτερο
Between constraint	Ανάμεσα- περιορισμός
Branching factor	Παράγοντας διακλάδωσης
Causal link	Αιτιολογικός σύνδεσμος
Classical planning	Κλασσικός σχεδιασμός
Constraint	Περιορισμός
Constructor	Συνάρτηση δημιουργίας/κατασκευής
Domain	Πεδίο σχεδιασμού
Effect	Επίδραση
Enumeration	Απαρίθμηση
Fact	Γεγονός
Flaw	Σφάλμα
Forward search	Εμπρόσθια/προς-τα-μπρος αναζήτηση
Fringe	Σύνορο (αναζήτησης)
Goal	Στόχος
Heuristic	Ευρετική
Hierarchical	Ιεραρχικό
Intelligent/rational agent	Έξυπνος πράκτορας
Literal	Λεκτικό
Ordering constraint	Περιορισμός σειράς
Plan	Πλάνο, σχέδιο
Plan space	Χώρος πλάνων
Plan space search	Αναζήτηση στο χώρο πλάνων
Planner	Σχεδιαστής, Καταστρωτής σχεδίου

Precedence constraint	Περιορισμοί προτεραιοτήτων
Precondition	Προϋπόθεση
Preprocessor	Προεπεξεργαστής
Proposition	Λογική έκφραση
Relaxation principle	Αρχή της χαλαρότητας
Satisfiability	Ικανοποιησιμότητας
Search space	Χώρος αναζήτησης
State	Κατάσταση
State space	Χώρος καταστάσεων
State space search	Αναζήτηση στο χώρο καταστάσεων
Subgoal	Υποστόχος
Threat	Απειλή
Union	Ένωση
Variable binding constraints	Περιορισμοί δέσμευσης μεταβλητών

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

AAAI	Association for the Advancement of Artificial Intelligence
AIIDE	Artificial Intelligence and Interactive Digital Entertainment Conference
FF	Fast Forward
GOAP	Goal-oriented action planning
HTN	Hierarchical task networks
PDDL	Plan domain description language
POP	Partial order planning
PSP	Plan space planning
SAT	Boolean Satisfiability
STRIPS	STanford Research Institute Problem Solver
TN	Τεχνητή Νοημοσύνη

ΠΑΡΑΡΤΗΜΑ Ι

Οδηγίες λήψης και εκτέλεσης

Ο κώδικας της βιβλιοθήκης iThink είναι διαθέσιμος στο δημόσιο αποθετήριο κώδικα και είναι προσπελάσιμος μέσω SVN ή HTTP. Βρίσκεται στο σύνδεσμο:

http://ithink-unity3d.googlecode.com/svn/

Παρέχονται και τα Unity project files (*.unity) για τη φόρτωση των παραδειγμάτων και εκτέλεσή τους απευθείας από το περιβάλλον Unity3D, το οποίο μπορείτε να μεταφορτώσετε από τον παρακάτω σύνδεσμο:

http://unity3d.com/unity/download/

Για να εκτελέσετε μέσω φυλλομετρητή διαθέσιμα Unity3D demos, χρειάζεστε τον Unity3D Web Player, τον οποίον μπορείτε να μεταφορτώσετε από τον παρακάτω σύνδεσμο:

http://unity3d.com/webplayer/

Οδηγίες για να βρείτε τα διαθέσιμα iThink demos για πειραματισμό βρίσκονται στο σύνδεσμο:

http://code.google.com/p/ithink-unity3d/source/browse/#svn/iThink/Samples/README.txt

ΠΑΡΑΡΤΗΜΑ ΙΙ

Πηγαίος κώδικας

iThinkAction.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/*! @class iThinkAction
    @brief iThinkAction escribes a base (abstract) Action and provides useful methods
concerning
           its status, initialization and functionality
    An action in iThink can be described by its \a name, its \a preconditions and its \a
effects.
    Its functions of interest are isApplicable() and applyEffect(), described below.
    If the user chooses the 1-parameter constructor, it is suggested to override the
initPreconditions() and initEffects() functions,
    and add the precondition and effect facts there.
*/

public class iThinkAction
{
    protected string name;                      /// The name of the action
    protected List<iThinkFact> preconditions;   /// The list of precondition facts
    protected List<iThinkFact> effects;         /// The list of effect facts

    public iThinkAction( string name ) { this.name = name; }
    public iThinkAction( string name, List<iThinkFact> preconditions, List<iThinkFact>
effects )
    {
        this.name = name;
        this.preconditions = new List<iThinkFact>( preconditions );
        this.preconditions = new List<iThinkFact>( effects );
    }

    /**
     * Returns a new iThinkState, based on \a State and applies the action's effects to
it
     * @param State The state to be effected
     * @returns A new iThinkState
     */
    public iThinkState applyEffects( iThinkState State )
    {
        iThinkState NewState = new iThinkState( State );

        foreach ( iThinkFact effect in this.effects )
            effect.applyFact( NewState );

        return NewState;
    }

    /**
     * Checks whether the action can be applied on iThinkState \a curState
     * @param curState The state to be checked
     * @returns A boolean value
     */
    public bool isApplicable( iThinkState curState )
    {
        int counter = 0;
        foreach ( iThinkFact fact in preconditions )
```

```
        {
            ///@todo Get facts of wanted type/name only
            foreach ( iThinkFact checkFact in curState.getFactList() )
            {
                if ( fact == checkFact )
                    counter++;
            }
        }
        if ( counter == preconditions.Count )
            return true;
        return false;
    }

    /// Used for initialization of the action's preconditions
    public virtual void initPreconditions() { preconditions = new List<iThinkFact>(); }
    /// Used for initialization of the action's effects
    public virtual void initEffects() { effects = new List<iThinkFact>(); }

    public string getName() { return name; }
    public List<iThinkFact> getEffects() { return effects; }

}
```

iThinkActionManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/*! @class iThinkActionManager
 *  @brief The ActionManager provides a list of actions that are usable by the agent via
iThinkBrain.
 *  By using iThinkActionSchemas, the action manager facilitates the creation of
available actions,
 *  which the user can access via \a getActions().
 */

public class iThinkActionManager
{
    protected List<iThinkAction> actionList;    /// List of available actions
    public iThinkActionSchemas schemaManager;   /// Schema manager to be used for action
generation

    public List<iThinkAction> getActions() { return actionList; }

    public iThinkActionManager()
    {
        actionList = new List<iThinkAction>();
        schemaManager = new iThinkActionSchemas();
    }

    /// This function will be called by the user whenever he wants to generate all
actions available to him.
    public void initActionList( GameObject agent, string[] actionSchemas,
List<GameObject> knownObjects, List<iThinkFact> factList )
    {
        List<iThinkAction> tempActionList = new List<iThinkAction>();
        foreach ( string schema in actionSchemas )
        {
            tempActionList.Clear();
            tempActionList = schemaManager.generateActions( knownObjects, factList,
schema, 2 );
            actionList.AddRange( tempActionList );
        }
    }
}
```

iThinkActionSchemas.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

#pragma warning disable 0660, 0661

/*! @class iThinkState
 * @brief iThinkState represents a state-node of the planning process
 * An iThinkState object is used in the planning process to describe a node of the
planning graph (search space).
 * It contains:
 * - its name
 * - a list of \a facts (literals)
 * - the evaluated \a cost of the state-node during the search process
 */

public class iThinkState
{
    protected List<iThinkFact> facts;   /// The list of facts which constitute the state
    protected int heuristicCost;        /// The total search cost up to this node, if
needed by the search algorithm
    protected string name;              /// The name of

    public iThinkState( string Name, List<iThinkFact> factList )
    {
        facts = new List<iThinkFact>( factList );
        heuristicCost = 0;
        name = Name;
    }

    public iThinkState( iThinkState state )
    {
        facts = new List<iThinkFact>( state.facts );
        heuristicCost = state.heuristicCost;
        name = state.name;
    }

    public string getName() { return name; }
    public void setName( string name ) { this.name = name; }
    public int getCost() { return heuristicCost; }
    public void setCost( int cost ) { heuristicCost = cost; }

    public List<iThinkFact> getFactList() { return facts; }

    /// If the fact isn't part of the list, it is inserted. Called by
iThinkFact::applyFact()
    public void addFact( iThinkFact fact )
    {
        foreach ( iThinkFact _fact in this.facts )
        {
            if ( _fact == fact )
                return;
        }
        facts.Add( fact );
    }

    /// If the fact is part of the list, it is deleted. Called by iThinkFact::applyFact()
    public void delFact( iThinkFact fact )
    {
        foreach ( iThinkFact _fact in this.facts )
        {
            if ( _fact == fact )
```

```
            {
                facts.Remove( _fact );
                break;
            }
        }
    }

    public static bool operator ==( iThinkState state1, iThinkState state2 )
    {
        if ( System.Object.ReferenceEquals( state1, state2 ) )
        {
            return true;
        }

        if ( (object)state1 == null || (object)state2 == null )
        {
            return false;
        }

        foreach ( iThinkFact fact1 in state1.facts )
        {
            bool check = false;

            foreach ( iThinkFact fact2 in state2.facts )
            {
                if ( fact1 == fact2 )
                    check = true;
            }

            if ( check == false )
                return false;
        }

        return true;
    }

    public static bool operator !=( iThinkState state1, iThinkState state2 )
    {
        return !( state1 == state2 );
    }

    public void debugPrint()
    {
        String msg = "[state] ";
        msg += this.name + " //";
        foreach ( var fact in facts )
            msg += " " + fact.getName();
        Debug.Log( msg );
    }
}
```

iThinkBrain.cs

```
/*!
 *  @mainpage iThink 0.1 documentation page
 *
 *  @section intro_sec Introduction
 *
 *  \subsection intro1 What is iThink?
 *  iThink is an AI library for Unity3D, implementing a STRIPS-like classical planning
system.
 *
 *  \section install Installation instructions
 *
 *  \section usage Usage
 *  \subsection usage1 Using the iThinkBrain
 *
 *  @todo iThinkBrain usage info
 *
 *  \subsection usage2 Using the iThinkPlanner
 *
 *  @todo iThinkPlanner usage info
 *
 *  \section license License
 *  GNU Lesser General Public License v3.0
 *
 *  \section credits Credits
 *  Code written by Vassilis Anasstassiou and Panagiotis Diamantopoulos
 */

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/** @class iThinkBrain
    @brief iThinkBrain contains all the needed components and sample interface to
implement an iThink agent.

    The iThink brain contains all planning-related systems and information.
    It provides the agent with:
    - a sensory system to populate the list of know GameObjects
    - a list of possible actions
    - its local knowledge (collection of facts (iThinkFact))
 */
public class iThinkBrain
{
    protected List<iThinkFact> factList;        /// A list of all the Facts known to the
agent.

    public iThinkState startState;              /// The first state before planning
begins.
    public iThinkState curState;                /// The current state.
    public iThinkState goalState;               /// The goal state.

    public iThinkActionManager ActionManager;   /// The list of available actions.
    public iThinkSensorySystem sensorySystem;   /// The system which updates the list of
known \a objects.
    public iThinkPlanner planner;               /// The iThink planner.

    public float lastUpdate;                    /// The last Time.time a check has been
performed.

    /// Getter for the list of objects known to the agent via his \a {sensory system}.
    public List<GameObject> getKnownObjects() { return sensorySystem.getKnownObjects(); }
```

```
    /// Getter for the list of facts known to the agent
    public List<iThinkFact> getKnownFacts() { return factList; }

    /// iThinkBrain constructor
    public iThinkBrain()
    {
        lastUpdate = Time.time;

        planner = new iThinkPlanner();
        factList = new List<iThinkFact>();
        sensorySystem = new iThinkSensorySystem();
    }
}
```

iThinkFact.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

#pragma warning disable 0660, 0661

/*! @class iThinkFact
    @brief iThinkFact describes a fact that can be used in the planning process and in
the description of actions.
    A fact describes a logic predicate. It consists of:
    - its \a name
    - the sign of the literal (whether it is a \a positive literal or not)
    - a list of its propositions (which are actually in-game \a GameObjects).
 */

public class iThinkFact
{
    protected List<GameObject> objects; /// The list of instances-parameters of the logic
literal.
    protected string name;              /// The name of the literal.
    protected bool positive;            /// Describes whether the literal is positive or
not.

    /// A new fact is positive by default
    public iThinkFact( string Name, params GameObject[] objs )
    {
        name = Name;
        positive = true;
        objects = new List<GameObject>();

        for ( int i = 0 ; i < objs.Length ; i++ )
            addObj( objs[i] );
    }

    /// If a new fact is negative, it must be specified with a second parameter in its
constructor, with value 'false'.
    public iThinkFact( string Name, bool pos, params GameObject[] objs )
    {
        name = Name;
        positive = pos;
        objects = new List<GameObject>();

        for ( int i = 0 ; i < objs.Length ; i++ )
            addObj( objs[i] );
    }

    public string getName() { return name; }
    public bool getPositive() { return positive; }

    public void addObj( GameObject obj ) { objects.Add( obj ); }
    public GameObject getObj( int index ) { return objects[index]; }
    public int getObjCount() { return objects.Count; }

    /// A positive iThinkFact is added to the \a State, and a negative one is removed
from it.
    public void applyFact( iThinkState State )
    {
        if ( this.positive == false )
            State.delFact( this );
        else
            State.addFact( this );
    }
```

```
    public static bool operator ==( iThinkFact fact1, iThinkFact fact2 )
    {
        if ( System.Object.ReferenceEquals( fact1, fact2 ) )
            return true;

        if ( (object)fact1 == null || (object)fact2 == null )
            return false;

        if ( !fact1.getName().Equals( fact2.getName() ) )
            return false;

        if ( fact1.objects.Count != fact2.objects.Count )
            return false;

        for ( int i = 0 ; i < fact1.objects.Count ; i++ )
        {
            if ( fact1.getObj( i ) != fact2.getObj( i ) )
                return false;
        }

        return true;
    }

    public static bool operator !=( iThinkFact fact1, iThinkFact fact2 )
    {
        return !( fact1 == fact2 );
    }
}
```

iThinkPlan.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/*! @class iThinkPlan
 *  @brief A plan is a list of actions paired to a state, describing an (incomplete)
plan.
 *  An iThinkPlan describes the sequence of "Actions" needed to proceed to a "State"
 *  When the State equals a goalstate, the Actions lists equals the Plan.
 */

public class iThinkPlan
{
    protected iThinkState State;
    protected List<iThinkAction> Actions;

    public iThinkPlan()
    {
        this.State = null;
        this.Actions = new List<iThinkAction>();
    }

    public iThinkPlan( iThinkState State )
    {
        this.State = new iThinkState( State );
        this.Actions = new List<iThinkAction>();
    }

    public iThinkPlan( iThinkState State, List<iThinkAction> Actions )
    {
        this.State = new iThinkState( State );
        this.Actions = new List<iThinkAction>( Actions );
    }

    public iThinkPlan( iThinkPlan Step )
    {
        this.State = new iThinkState( Step.State );
        this.Actions = new List<iThinkAction>( Step.Actions );
    }

    public iThinkState getState() { return State; }
    public void setState( iThinkState NewState ) { State = NewState; }

    public List<iThinkAction> getPlanActions() { return Actions; }
    public int getActionCount() { return Actions.Count; }

    public bool hasPlan() { return Actions.Count != 0; }
    public void setPlan( iThinkPlan plan ) { Actions = new List<iThinkAction>(
plan.getPlanActions() ); }

    public void debugPrintPlan()
    {
        String msg = "[PrintPlan()]";
        foreach ( var action in Actions )
        { msg += " -> " + action.getName(); }
        Debug.Log( msg );
    }
}
```

iThinkPlanner.cs

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/*! @class iThinkPlanner
 *  @brief iThinkPlanner performs the planning process
 *  By supplying an initial state, a goal state, and an iThinkActionManager object,
iThinkPlanner
 *  performs planning and searches for the first valid plan.
 */

public class iThinkPlanner
{
    protected iThinkPlan Plan;
    protected List<iThinkPlan> _OpenStates;
    protected List<iThinkState> _VisitedStates;

    public iThinkPlanner()
    {
        Plan = new iThinkPlan();
        _OpenStates = new List<iThinkPlan>();
        _VisitedStates = new List<iThinkState>();
    }

    public iThinkPlan getPlan() { return Plan; }

    /// Just an interface to call forwardSearch with BestFS search method
    public bool forwardSearch( iThinkState InitialState, iThinkState GoalState,
iThinkActionManager ActionManager )
    {
        return forwardSearch( InitialState, GoalState, ActionManager, 1 );
    }

    /// The function performing planning using Forward Search and the specified \a method
    public bool forwardSearch( iThinkState InitialState, iThinkState GoalState,
iThinkActionManager ActionManager, int Method )
    {
        iThinkPlan ReturnVal;

        _OpenStates.Clear();
        _VisitedStates.Clear();

        iThinkPlan step = new iThinkPlan( InitialState );
        _OpenStates.Add( step );
        _VisitedStates.Add( step.getState() );

        switch ( Method )
        {
            case 0:
                ReturnVal = depthFS( GoalState, ActionManager, _OpenStates,
_VisitedStates );
                break;
            case 1:
                ReturnVal = bestFS( GoalState, ActionManager, _OpenStates, _VisitedStates
);
                break;
            case 2:
                ReturnVal = breadthFS( GoalState, ActionManager, _OpenStates,
_VisitedStates );
                break;
            default:
```

```
                ReturnVal = new iThinkPlan();
                break;
        }

        if ( ReturnVal == null )
            return false;
        else if ( ReturnVal.hasPlan() )
            return true;
        return false;
    }

    /// Depth-First Search
    public iThinkPlan depthFS( iThinkState GoalState, iThinkActionManager ActionManager,
List<iThinkPlan> OpenStates, List<iThinkState> VisitedStates )
    {
        int it = 0;
        iThinkPlan curStep, nextStep;
        iThinkState CurrentState;

        while ( OpenStates.Count != 0 )
        {
            //Debug.LogWarning( "Iteration #" + it );
            List<iThinkAction> applicableActions = new List<iThinkAction>();

            curStep = new iThinkPlan( OpenStates[0] );
            CurrentState = OpenStates[0].getState();
            OpenStates.RemoveAt( 0 );

            CurrentState.debugPrint();

            applicableActions = getApplicable( CurrentState, ActionManager.getActions()
);


            foreach ( iThinkAction action in applicableActions )
            {
                bool found = false;

                // todo: Add "statnode" for statistics retrieval
                nextStep = progress( curStep, action );

                if ( compareStates( nextStep.getState(), GoalState ) )
                {
                    Debug.Log( "Found Plan (DepthFS)" );
                    Plan.setPlan( nextStep );
                    return Plan;
                }

                foreach ( iThinkState state in VisitedStates )
                {
                    if ( state == nextStep.getState() )
                    {
                        found = true;
                        break;
                    }
                }

                if ( found == false )
                {
                    OpenStates.Insert( 0, nextStep );
                    VisitedStates.Add( nextStep.getState() );
                }
            }

            ++it;
        }
```

```
        Debug.Log( "Didn't find Plan (DepthFS)" );
        return null;
    }

    /// Breadth-First Search
    public iThinkPlan breadthFS( iThinkState GoalState, iThinkActionManager
ActionManager, List<iThinkPlan> OpenStates, List<iThinkState> VisitedStates )
    {
        int it = 0;
        iThinkPlan curStep, nextStep;
        iThinkState CurrentState;

        while ( OpenStates.Count != 0 )
        {
            //Debug.LogWarning( "Iteration #" + it );
            List<iThinkAction> applicableActions = new List<iThinkAction>();

            curStep = new iThinkPlan( OpenStates[0] );
            CurrentState = OpenStates[0].getState();
            OpenStates.RemoveAt( 0 );

            CurrentState.debugPrint();

            applicableActions = getApplicable( CurrentState, ActionManager.getActions()
);

            foreach ( iThinkAction action in applicableActions )
            {
                bool found = false;

                // todo: Add "statnode" for statistics retrieval
                nextStep = progress( curStep, action );

                if ( compareStates( nextStep.getState(), GoalState ) )
                {
                    Debug.Log( "Found Plan (BreadthFS)" );
                    Plan.setPlan( nextStep );
                    return Plan;
                }

                foreach ( iThinkState state in VisitedStates )
                {
                    if ( state == nextStep.getState() )
                    {
                        found = true;
                        break;
                    }
                }

                if ( found == false )
                {
                    OpenStates.Add( nextStep );
                    VisitedStates.Add( nextStep.getState() );
                }
            }

            ++it;
        }
        Debug.Log( "Didn't find Plan (BreadthFS)" );
        return null;
    }

    /// Best-First Search, which uses a Manhattan-distance heuristic (number of not-yet
satisfied goalstate facts)
```

```
    public iThinkPlan bestFS( iThinkState GoalState, iThinkActionManager ActionManager,
List<iThinkPlan> OpenStates, List<iThinkState> VisitedStates )
    {
        int it = 0;
        iThinkPlan curStep, nextStep;
        iThinkState CurrentState;

        while ( OpenStates.Count != 0 )
        {
            //Debug.LogWarning( "Iteration #" + it );
            List<iThinkAction> applicableActions = new List<iThinkAction>();
            List<iThinkPlan> stateList = new List<iThinkPlan>();

            curStep = new iThinkPlan( OpenStates[0] );
            CurrentState = OpenStates[0].getState();
            OpenStates.RemoveAt( 0 );

            applicableActions = getApplicable( CurrentState, ActionManager.getActions()
);


            foreach ( iThinkAction action in applicableActions )
            {
                bool found = false;
                // todo: Add "statnode" for statistics retrieval
                nextStep = progress( curStep, action );

                if ( compareStates( nextStep.getState(), GoalState ) )
                {
                    Debug.Log( "Found Plan (BestFS)" );
                    Plan.setPlan( nextStep );
                    return Plan;
                }

                foreach ( iThinkState state in VisitedStates )
                {
                    if ( state == nextStep.getState() )
                    {
                        found = true;
                        break;
                    }
                }

                if ( found == false )
                {
                    int Cost = hFunction( nextStep.getState(), GoalState );
                    nextStep.getState().setCost( Cost );
                    stateList.Add( nextStep );
                    VisitedStates.Add( nextStep.getState() );
                }

            }

            OpenStates.AddRange( stateList );
            OpenStates.Sort( delegate( iThinkPlan obj1, iThinkPlan obj2 )
                             {
                                 if ( obj1.getState().getCost() ==
obj2.getState().getCost() )
                                     return 0;
                                 else if ( obj1.getState().getCost() >
obj2.getState().getCost() )
                                     return -1;
                                 else
                                     return 1;
                             }
                            );
```

```
            ++it;
        }
        Debug.Log( "Didn't find plan (BestFS)" );
        return null;
    }
    /// Manhattan-distance heuristic computation for BestFS()
    private int hFunction( iThinkState nextState, iThinkState GoalState )
    {
        int counter = 0;
        foreach ( iThinkFact fact in nextState.getFactList() )
        {
            foreach ( iThinkFact goalFact in GoalState.getFactList() )
            {
                if ( fact == goalFact )
                {
                    counter++;
                    break;
                }
            }
        }
        return counter;
    }

    /// Generates the next plan step after the application of \a Action's effects.
    public iThinkPlan progress( iThinkPlan Step, iThinkAction Action )
    {
        iThinkPlan NewStep = new iThinkPlan( Step );
        List<iThinkAction> curActions;

        curActions = NewStep.getPlanActions();
        curActions.Add( Action );

        NewStep.setState( Action.applyEffects( Step.getState() ) );

        return NewStep;
    }

    /// Finds all actions applicable to the current \a State from the collection of
available \a Actions
    public List<iThinkAction> getApplicable( iThinkState State, List<iThinkAction>
Actions )
    {
        List<iThinkAction> ApplicableActions = new List<iThinkAction>();

        foreach ( iThinkAction action in Actions )
        {
            if ( action == null )
                break;

            if ( action.isApplicable( State ) )
                ApplicableActions.Add( action );
        }

        return ApplicableActions;
    }

    /// Checks if goalState is a subset of curState
    private bool compareStates( iThinkState curState, iThinkState goalState )
    {
        int counter = 0;
        foreach ( iThinkFact fact in goalState.getFactList() )
        {
            foreach ( iThinkFact check in curState.getFactList() )
            {
                if ( check == null )
```

```
                    return false;
                else if ( check == fact )
                    counter++;
            }
        }
        if ( counter == goalState.getFactList().Count )
            return true;
        return false;
    }
}
```

iThinkSensorySystem.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/*! @class iThinkSensorySystem
 * @brief A system implementing GameObject detection via Tags
 * Implements a basic system for declaring sensors usable by iThink agents.
 * Currently, it only provides two kinds of sensors:
 * - an active sensor \a (OmniscientUpdate) which returns all objects of specified \a
Tags.
 * - an active sensor \a (ProximityUpdate) which returns all objects within a specified
\a radius
 *
 * It's up to the user to extend the class with more kinds of sensors.
 */

public class iThinkSensorySystem
{
    protected List<GameObject> knownObjects;   /// A list of all the GameObjects sensed
by the agent

    public iThinkSensorySystem() { knownObjects = new List<GameObject>(); }

    public List<GameObject> getKnownObjects() { return knownObjects; }

    /// Performs a search for GameObjects within a specified \a radius from the agent
    public void ProximityUpdate( GameObject agent, UInt32 radius )
    {
        LayerMask layerMask = 1 << LayerMask.NameToLayer( "GamePart" );
        Collider[] interactiveObjs = Physics.OverlapSphere( agent.transform.position,
radius, layerMask.value );

        foreach ( Collider col in interactiveObjs )
            knownObjects.Add( col.gameObject );
    }

    /// Performs a universal search in the whole scene for GameObjects of specified \a
Tags
    public void OmniscientUpdate( GameObject agent, List<String> Tags )
    {
        GameObject[] objects;

        foreach ( String tag in Tags )
        {
            objects = GameObject.FindGameObjectsWithTag( tag );

            foreach ( GameObject obj in objects )
                knownObjects.Add( obj );
        }
    }
}
```

iThinkState.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

#pragma warning disable 0660, 0661

/*! @class iThinkState
 * @brief iThinkState represents a state-node of the planning process
 * An iThinkState object is used in the planning process to describe a node of the
planning graph (search space).
 * It contains:
 * - its name
 * - a list of \a facts (literals)
 * - the evaluated \a cost of the state-node during the search process
 */

public class iThinkState
{
    protected List<iThinkFact> facts;   /// The list of facts which constitute the state
    protected int heuristicCost;        /// The total search cost up to this node, if
needed by the search algorithm
    protected string name;              /// The name of

    public iThinkState( string Name, List<iThinkFact> factList )
    {
        facts = new List<iThinkFact>( factList );
        heuristicCost = 0;
        name = Name;
    }

    public iThinkState( iThinkState state )
    {
        facts = new List<iThinkFact>( state.facts );
        heuristicCost = state.heuristicCost;
        name = state.name;
    }

    public string getName() { return name; }
    public void setName( string name ) { this.name = name; }
    public int getCost() { return heuristicCost; }
    public void setCost( int cost ) { heuristicCost = cost; }

    public List<iThinkFact> getFactList() { return facts; }

    /// If the fact isn't part of the list, it is inserted. Called by
iThinkFact::applyFact()
    public void addFact( iThinkFact fact )
    {
        foreach ( iThinkFact _fact in this.facts )
        {
            if ( _fact == fact )
                return;
        }
        facts.Add( fact );
    }

    /// If the fact is part of the list, it is deleted. Called by iThinkFact::applyFact()
    public void delFact( iThinkFact fact )
    {
        foreach ( iThinkFact _fact in this.facts )
        {
            if ( _fact == fact )
```

```
            {
                facts.Remove( _fact );
                break;
            }
        }
    }

    public static bool operator ==( iThinkState state1, iThinkState state2 )
    {
        if ( System.Object.ReferenceEquals( state1, state2 ) )
        {
            return true;
        }

        if ( (object)state1 == null || (object)state2 == null )
        {
            return false;
        }

        foreach ( iThinkFact fact1 in state1.facts )
        {
            bool check = false;

            foreach ( iThinkFact fact2 in state2.facts )
            {
                if ( fact1 == fact2 )
                    check = true;
            }

            if ( check == false )
                return false;
        }

        return true;
    }

    public static bool operator !=( iThinkState state1, iThinkState state2 )
    {
        return !( state1 == state2 );
    }

    public void debugPrint()
    {
        String msg = "[state] ";
        msg += this.name + " //";
        foreach ( var fact in facts )
            msg += " " + fact.getName();
        Debug.Log( msg );
    }
}
```

ΠΑΡΑΡΤΗΜΑ III

Τεκμηρίωση

Σχόλια πηγαίου κώδικα είναι διαθέσιμα στο παράρτημα II.

Μπορείτε να βρείτε τη πλήρη τεκμηρίωση σε εναλλακτικές μορφές (που παράχθηκαν μέσω του συστήματος Doxygen) στο σύνδεσμο:
http://code.google.com/p/ithink-unity3d/source/browse/#svn/Doc

ΑΝΑΦΟΡΕΣ

[1] Fikes, R. & Nilsson, N. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. Προσπέλαση από: http://www.ai.sri.com/pubs/files/tn043r-fikes71.pdf

[2] Orkin, J. (2005). Agent Architecture Considerations for Real-Time Planning in Games. Monolith Productions, Inc. Προσπέλαση από: http://web.media.mit.edu/~jorkin/aiide05OrkinJ.pdf

[3] Blum, A. L., & Furst, M. L. (1997). Fast planning through planning graph analysis. Artificial Intelligence, 90 (1-2), 281-300. Elsevier. Προσπέλαση από: http://www.cs.cmu.edu/~avrim/Papers/graphplan.pdf

[4] Kautz, H., & Selman, B. (1992). Planning as Satisfiability. Proceedings of the European Conference on Artificial Intelligence ECAI (Vol. 54, pp. 359-363). John Wiley & Sons, Inc. Προσπέλαση από: http://www.plg.inf.uc3m.es/~pad/06-07/papers/kautz92planning.pdf

[5] Erol, K., Hendler, J., & Nau, D. S. (1994). HTN Planning: Complexity and Expressivity. Proceedings of the twelfth national conference on Artificial intelligence vol 2 (Vol. 2, pp. 1123-1128). AAAI Press. Προσπέλαση από: http://www.cs.umd.edu/~nau/papers/erol1994htn.pdf

[6] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., et al. (1998). PDDL - The Planning Domain Definition Language. Annals of Physics. Yale University. Προσπέλαση από: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212

[7] Hoffmann, J. (2001). FF: The Fast-Forward Planning System. AI Magazine, 22(3), 57-62. Προσπέλαση από: http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1572

[8] Kautz, H., & Selman, B. (1998). BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. AIPS98 Workshop on Planning as Combinatorial Search (Vol. 58260, pp. 58-60). Working Notes of the AIPS-98 Workshop on Planning as Combinatorial Search. Προσπελάστηκε από: http://www.cs.rochester.edu/~kautz/papers/aips98-kautz.ps

[9] Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D, & Yaman, F. (2003). SHOP2 : An HTN Planning System. Systems Research, 20(1), 379-404. AI Access Foundation. Προσπέλαση από: http://www.cs.utexas.edu/~chiu/papers/Nau03shop2.pdf

[10] Vassos, S., Papakonstantinou, M. (2011). The SimpleFPS Planning Domain: A PDDL Benchmark for Proactive NPCs. Proceedings of the Non-Player Character AI workshop (NPCAI-2011) of the Artificial Intelligence & Interactive Digital Entertainment (AIIDE-2011) Conference, Stanford CA. Προσπέλαση από: http://stavros.lostre.org/files/Vassos11SimpleFPS.pdf

[11] Russell, S., & Norvig, P . Τεχνητή Νοημοσύνη, μια σύγχρονη προσέγγιση, 2η εκτύπωση (σκληρό εξώφυλλο). Εκδόσεις Κλειδάριθμος. ISBN: 960-209-873-2. Πρωτότυπο έργο: (2002). Prentice Hall.

[12] Ι. Βλαχάβας, Π. Κεφαλάς, Ν. Βασιλειάδης, Φ. Κόκκορας, Η. Σακελλαρίου. (2011). Τεχνητή Νοημοσύνη – Γ' Έκδοση. ISBN: 978-960-8396-64-7, Εκδόσεις Πανεπιστημίου Μακεδονίας

[13] Ghallabm, M., Nau, D., Traverso, P. (2004). Automated Planning Theory & Practice. ISBN: 1558608567

[14] Millington, I., Funge, J. (2009). Artificial Intelligence for Games, 2nd Edition. ISBN: 9780123747310

[15] LaValle, S. M. (2006). Planning Algorithms. ISBN: 0521862051

[16] Bjarnolf, P. (2008). Threat Analysis Using Goal-Oriented Action Planning Planning in the Light of Information Fusion. Information Fusion. University of Skövde, School of Humanities and Informatics. Προσπέλαση από: http://his.diva-portal.org/smash/get/diva2:2228/FULLTEXT01

[17] Orkin, J. (2006). Three states and a plan: the AI of FEAR. Game Developers Conference (Vol. 2006, pp. 1-18). Citeseer. Προσπέλαση από: http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf

[18] Buro, M. (2004). Call for AI Research in RTS Games. In D. Fu, S. Henke, & J. Orkin (Eds.), Computing (pp. 2-4). Προσπέλαση από: http://www.aaai.org/Papers/Workshops/2004/WS-04-04/WS04-04-028.pdf

[19] Wu, J., Kayanam, R., & Givan, R. (2008). Planning using stochastic enforced hill-climbing. IPPC08 (pp. 396-403). Unavailable. Προσπέλαση από: http://ippc-2008.loria.fr/wiki/images/9/94/Team6-SEH.pdf

[20] Kelly, J.-P., Botea, A., & Koenig, S. (2008). Offline Planning with Heirarchical Task Netwroks in Video Games. AIIDE 2008 (Vol. 90, pp. 60-65). www.cs.ucf.edu/~gitars/cap6671/Papers/AIIDE08-010.pdf

[21] Bartheye, O. (2010). Real-Time Planning for Video-Games: A Purpose for PDDL. ICAPS 2010 Planning in Games Workshop. Προσπέλαση από: http://skatgame.net/mburo/icaps2010-pg/ICAPS-PG.2010.1.bartheye.pdf