ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΤΜΗΜΑ ΜΑΘΗΜΑΤΙΚΩΝ
ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ
"ΘΕΩΡΗΤΙΚΗ ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΘΕΩΡΙΑ ΣΥΣΤΗΜΑΤΩΝ ΚΑΙ ΕΛΕΓΧΟΥ"

ΑΝΑΠΤΥΞΗ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ ΓΙΑ ΠΑΙΧΝΙΔΙ ΣΤΡΑΤΗΓΙΚΗΣ ΠΡΑΓΜΑΤΙΚΟΥ ΧΡΟΝΟΥ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Γιανταμίδης

Επιβλέπων: Α. Ι. Βαρδουλάκης
Καθηγητής Α.Π.Θ.

Θεσσαλονίκη, Φεβρουάριος 2012

ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΤΜΗΜΑ ΜΑΘΗΜΑΤΙΚΩΝ
ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ
"ΘΕΩΡΗΤΙΚΗ ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΘΕΩΡΙΑ ΣΥΣΤΗΜΑΤΩΝ ΚΑΙ ΕΛΕΓΧΟΥ"

ΑΝΑΠΤΥΞΗ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ ΓΙΑ ΠΑΙΧΝΙΔΙ ΣΤΡΑΤΗΓΙΚΗΣ ΠΡΑΓΜΑΤΙΚΟΥ ΧΡΟΝΟΥ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Γιανταμίδης

Επιβλέπων: Α. Ι. Βαρδουλάκης
Καθηγητής Α.Π.Θ.

Εγκρίθηκε από την τριμελή επιτροπή την.....

.....	.....	.....
Ν. Καραμπετάκης	Ε. Αντωνίου	Α. Ι. Βαρδουλάκης
Επίκουρος Καθηγητής Α.Π.Θ.	Επίκουρος Καθηγητής – ΑΤΕΙΘ	Καθηγητής Α.Π.Θ.

Θεσσαλονίκη, Φεβρουάριος 2012

.....

Γεώργιος Γιανταμίδης
Πτυχιούχος Μαθηματικός Α.Π.Θ.

Copyright © Γεώργιος Γιανταμίδης, 2012
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς το συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν το συγγραφέα και δεν πρέπει να ερμηνευτεί ότι εκφράζουν τις επίσημες θέσεις του Α.Π.Θ.

ΠΕΡΙΛΗΨΗ

Στην παρούσα εργασία θα δούμε πως μπορούμε να φτιάξουμε ένα bot για το StarCraft, δηλαδή ένα πρόγραμμα το οποίο μπορεί να παίζει StarCraft στη θέση μας (το StarCraft είναι ένα πολύ δημοφιλές παιχνίδι στρατηγικής πραγματικού χρόνου για Η/Υ). Αρχικά, θα πούμε λίγα λόγια για τη διαδικασία ανάπτυξης του bot, έπειτα λίγα λόγια για το παιχνίδι και στη συνέχεια θα δούμε αναλυτικά τις τεχνικές και τους αλγορίθμους που χρησιμοποιήθηκαν, καθώς και πώς ακριβώς χρησιμοποιήθηκαν. Τέλος, παρουσιάζουμε τον κώδικα του προγράμματος, αρχικά περιγράφοντας συνοπτικά το ρόλο κάθε κομματιού και έπειτα αναλύοντας κάποιες use-cases (περιπτώσεις χρήσης), δηλαδή βλέποντας βήμα προς βήμα "τι συμβαίνει όταν θέλω να κάνω αυτό;".

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

Τεχνητή Νοημοσύνη, StarCraft, bot

ABSTRACT

In the present project, we will see how one can write a bot for StarCraft, that is, a program that can play StarCraft on its own (StarCraft is a very popular real time strategy game for PCs). At first, we will talk a bit the bot developing process, then about the game and afterwards we will examine the techniques and algorithms used, as well as the exact way they were used. Finally, the code of the program is presented. At first, the role of each part is described briefly and then a couple of use-cases are analyzed, that is, we examine step by step “what happens when I want to do this?”.

KEYWORDS

Artificial Intelligence, StarCraft, bot

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγικά

Όταν έφτασε η ώρα να διαλέξω θέμα για διπλωματική εργασία, σκέφτηκα ότι ήθελα να κάνω κάτι:

(1) Που θα περιλαμβάνει προγραμματισμό.

(2) Από το οποίο θα μάθω καινούρια πράγματα
που θα μου χρησιμεύσουν και αργότερα.

(3) Το οποίο θα έχει πρακτική αξία πέρα από το
γεγονός ότι θα αποτελεί διπλωματική εργασία.

Και έτσι αποφάσισα να γράψω ένα bot για το StarCraft, δηλαδή ένα πρόγραμμα που θα παίζει StarCraft στη θέση μου. Επικοινώνησα με τον κύριο Βαρδουλάκη, ο οποίος αμέσως συμφώνησε με την ιδέα και με προέτρεψε να προχωρήσω.

Όταν άρχισα να δουλεύω είχα πολύ μεγάλες προσδοκίες για το τελικό αποτέλεσμα. Ήθελα να φτιάξω κάτι που θα μπορούσε να νικήσει οποιοδήποτε άνθρωπο ή άλλο bot σε μια αναμέτρηση 1 vs 1. Και φαντάστηκα ότι για να το πετύχω αυτό θα έπρεπε να χρησιμοποιήσω όσο γίνεται πιο πολλές διαφορετικές τεχνικές. Αρκετές από τις τεχνικές που είχα στο μυαλό μου, όμως, είτε ήταν δύσκολο να υλοποιηθούν είτε δεν προσέθεταν πολύ στην αποτελεσματικότητα του bot (ή, κάποιες φορές, και τα δύο). Γι' αυτό άλλαξα λίγο τον τελικό μου στόχο. Αποφάσισα ότι θα ήταν καλύτερα σε πρώτη φάση να γράψω κάτι το οποίο θα μπορεί να νικάει το ενσωματωμένο AI του παιχνιδιού με σχετική ευκολία, και που θα είναι εύκολα επεκτάσιμο με νέες στρατηγικές. Και αυτό είναι αυτό που θα παρουσιάσω εδώ.

Δε θα σταματήσω εδώ όμως. Τώρα που έχω κάτι πολύ καλό σαν βάση, θα συνεχίσω να προσθέτω στοιχεία σιγά σιγά, και όταν φτάσω σε ένα αρκετά ικανοποιητικό επίπεδο σκοπεύω να πάρω μέρος σε έναν από τους διαγωνισμούς για StarCraft bots που διεξάγονται κάθε χρόνο.

Τα εργαλεία που χρησιμοποιήθηκαν ήταν τα εξής: BWAPI, Chaos Launcher, Visual C++ 2008 Express Edition. Το πρώτο είναι μία ανοικτή βιβλιοθήκη που επιτρέπει σε κώδικα C++ να αλληλεπιδρά με το StarCraft. Επιτρέπει στον προγραμματιστή να έχει πρόσβαση σε οποιαδήποτε πληροφορία της κατάστασης του παιχνιδιού μπορεί να έχει ένας παίκτης, καθώς και να μεταβάλει την κατάσταση του παιχνιδιού με οποιοδήποτε τρόπο μπορεί να το κάνει ένας παίκτης. Το δεύτερο είναι

ένα πρόγραμμα απαραίτητο για την επικοινωνία του bot με το StarCraft. Το τελευταίο είναι το προγραμματιστικό περιβάλλον που χρησιμοποιήθηκε για τη συγγραφή του bot.

2. Λίγα λόγια για το παιχνίδι [10]

Το StarCraft είναι ένα παιχνίδι στρατηγικής πραγματικού χρόνου και αναπτύχθηκε από τη Blizzard Entertainment. Το πρώτο παιχνίδι της σειράς κυκλοφόρησε το Μάρτιο του 1998. Μέχρι το Φεβρουάριο του 2009, είχαν πουληθεί περισσότερα από 11 εκατομμύρια αντίτυπα παγκοσμίως, καθιστώντας το ένα από τα best-selling παιχνίδια για ηλεκτρονικό υπολογιστή. Πρόσφατα (τον Ιούλιο του 2010) κυκλοφόρησε και το δεύτερο παιχνίδι της σειράς, το οποίο επίσης γνώρισε μεγάλη επιτυχία. Το StarCraft είναι ιδιαίτερα δημοφιλές στη Νότια Κορέα, όπου παίκτες και ομάδες μπορούν να συμμετέχουν σε διαγωνισμούς με μεγάλα χρηματικά έπαθλα.

Η ιστορία του παιχνιδιού διαδραματίζεται στον 26^{ο} αιώνα, σε ένα μακρινό από μας μέρος του γαλαξία, όπου τρεις φυλές μάχονται για κυριαρχία: (1) οι Terrans, γήινοι που έχουν εξοριστεί από τον πλανήτη, (2) Οι Zerg, μια φυλή εντομοειδών εξωγήινων που αναζητά γενετική τελειότητα αφομοιώνοντας άλλα είδη ζωής, και (3) οι Protoss, μια ανθρωποειδής φυλή με πολύ ανεπτυγμένη τεχνολογία και τηλεπαθητικές ικανότητες, που προσπαθεί να σώσει τον πολιτισμό της από την απειλή των Zerg.

Η απόφαση της Blizzard να χρησιμοποιήσει τρεις διαφορετικές φυλές στο StarCraft ήταν τότε επαναστατική για το συγκεκριμένο είδος παιχνιδιού. Κάθε φυλή έχει διαφορετικά είδη μονάδων από τις υπόλοιπες και απαιτεί διαφορετική τακτική από τον παίκτη. Οι Protoss γενικά έχουν πρόσβαση σε δυνατές μονάδες που όμως είναι σχετικά ακριβό να παραχθούν και έτσι αναγκάζουν τον παίκτη να ακολουθήσει στρατηγική που δίνει βάρος στην ποιότητα του στρατού και όχι στην ποσότητα. Από την άλλη, οι Zerg διαθέτουν κτήρια που επιτρέπουν πολύ γρήγορη και φθηνή παραγωγή μονάδων, αλλά κάθε μονάδα είναι σχετικά εύθραυστη. Οι Terrans βρίσκονται κάπου στη μέση. Πάντως, παρά τη μεγάλη ποικιλία μονάδων που συναντάμε στο παιχνίδι, η Blizzard έχει φροντίσει ώστε καμία φυλή να μην έχει πλεονέκτημα απέναντι σε κάποια άλλη.

Στο StarCraft, ο παίκτης στηρίζεται σε δυο είδη υλικών πόρων για να αναπτυχθεί: minerals και vespene gas. Το πρώτο είναι απαραίτητο για σχεδόν όλα τα κτήρια και τις μονάδες, και το δεύτερο είναι απαραίτητο για τα πιο ανεπτυγμένα από αυτά. Επιπλέον, ο αριθμός των μονάδων που μπορεί να έχει ο παίκτης περιορίζεται

από τον αριθμό των ‘προμηθειών’ που διαθέτει την εκάστοτε χρονική στιγμή. Επιτρέπεται, λοιπόν, στον παίκτη να χτίσει καινούριες μονάδες μόνο αν διαθέτει τις απαραίτητες προμήθειες, οι οποίες συνήθως μπορούν να αυξηθούν χτίζοντας απλά ένα (συγκεκριμένου είδους) κτήριο.

Η ικανή και αναγκαία συνθήκη για τη νίκη είναι η καταστροφή όλων των κτηρίων όλων των αντιπάλων.

3. παρουσίαση των τεχνικών που χρησιμοποιούνται

-> Χάρτες επιρροής (influence maps ή influence mapping)
-> Ευφυΐα σμήνους (swarm intelligence)
-> Finite state machines
-> A* (προφέρεται «έι σταρ»)
-> Γενετικοί αλγόριθμοι (genetic algorithms)

3.1 Χάρτες επιρροής (influence maps) [2, 3, 4]

Αυτή είναι μια σχετικά απλή τεχνική, η οποία δεν έχει μεγάλη αξία από μόνη της, αλλά είναι πάρα πολύ χρήσιμη όταν συνδυάζεται με τις άλλες τεχνικές (κάτι που κάνω συχνά όπως θα δείτε), γι’ αυτό και θεώρησα καλύτερο να την παρουσιάσω πρώτη.

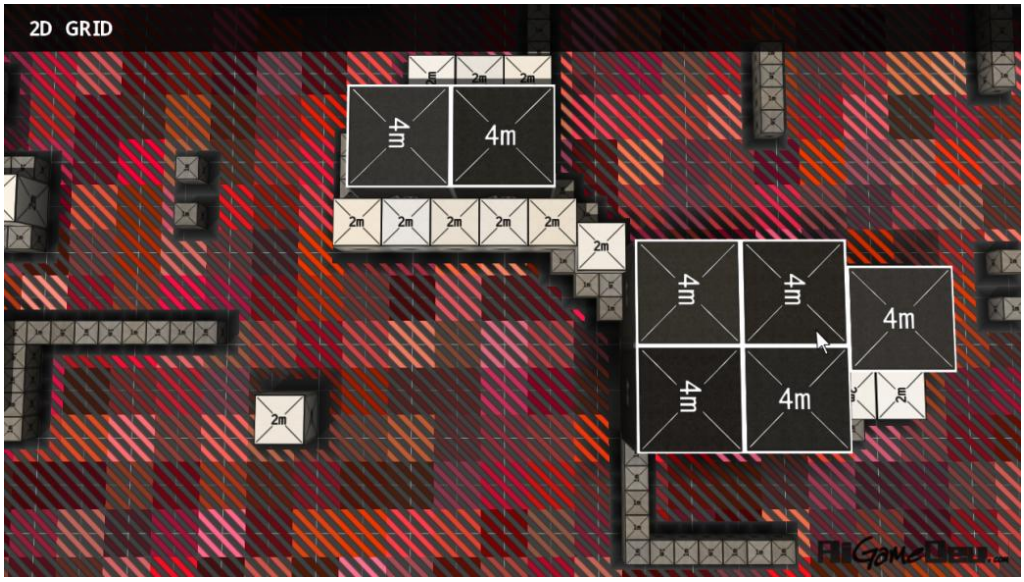
Ένα από τα πιο δύσκολα προβλήματα που αντιμετωπίζει κανείς όταν ασχολείται με τεχνητή νοημοσύνη είναι η μετατροπή του σχετικά περίπλοκου κόσμου του παιχνιδιού σε ένα απλό σύνολο δεδομένων το οποίο μπορεί εύκολα να αναλύσει ώστε να μπορεί να πάρει αποφάσεις. Οι χάρτες επιρροής είναι μια πολύ συχνά χρησιμοποιούμενη λύση για το συγκεκριμένο πρόβλημα.

Η βασική ιδέα πίσω από αυτήν την τεχνική είναι ότι κάθε αντικείμενο του κόσμου του παιχνιδιού επηρεάζει με κάποιον τρόπο τη θέση στην οποία βρίσκεται και αυτή η επιρροή εξαπλώνεται και στις γειτονικές θέσεις (βέβαια, όσο απομακρυνόμαστε από την αρχική θέση η τιμή της επιρροής αλλάζει – συνήθως μειώνεται, αλλά όχι πάντα). Αν υπολογίσουμε την επιρροή που ασκεί κάθε αντικείμενο στο χώρο και αθροίσουμε αυτές τις επιρροές έχουμε ένα χάρτη επιρροής.

Υπάρχουν διάφοροι τρόποι για την αναπαράσταση ενός χάρτη επιρροής. Ενδεικτικά αναφέρουμε τους εξής:

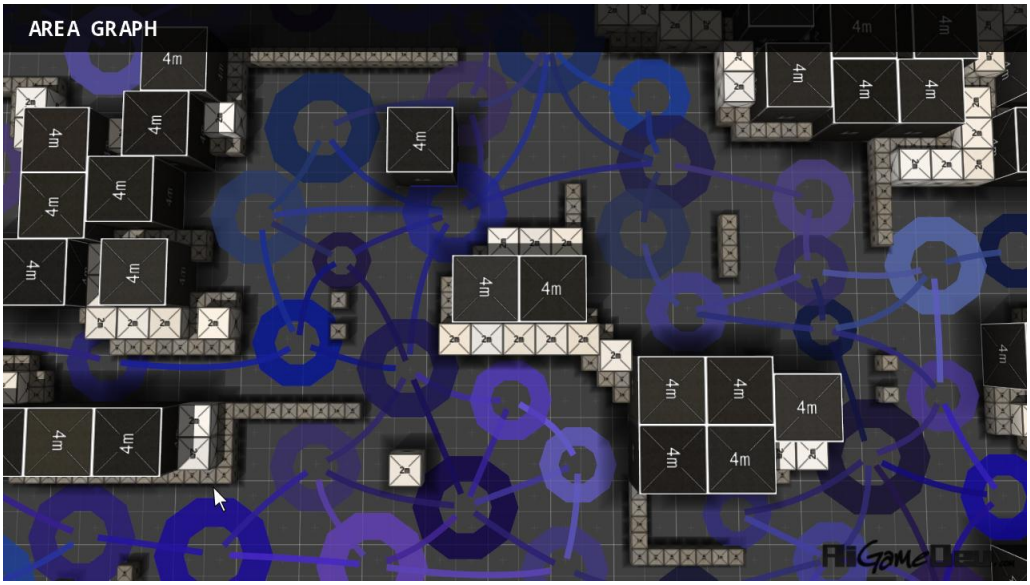
(α) δισδιάστατα πλέγματα

Αυτός είναι ο πιο συνηθισμένος τρόπος αναπαράστασης και είναι και αυτός που χρησιμοποιώ εδώ. Είναι πολύ απλός στην υλοποίηση και καθιστά την επεξεργασία του χάρτη επιρροής πολύ γρήγορη. Η ανάλυση (resolution) του πλέγματος επιλέγεται να είναι αρκετά μεγάλη για να μη χάνονται σημαντικές λεπτομέρειες, αλλά όχι πολύ μεγάλη ώστε να γίνεται σπατάλη μνήμης.



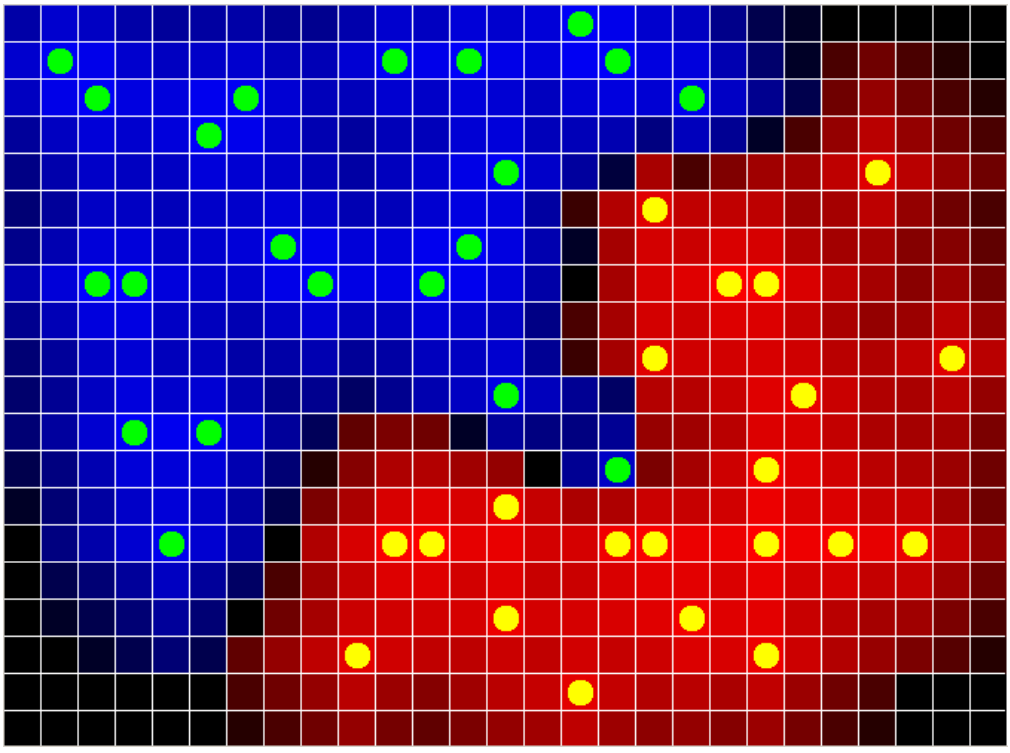
(β) γράφοι περιοχών

Αυτός είναι ένας άλλος τρόπος αναπαράστασης. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι σε πολλά παιχνίδια ούτως ή άλλως χρησιμοποιούνται γράφοι περιοχών για τη βοήθεια του προβλήματος του pathfinding και επομένως η ίδια δομή μπορεί με ελαφριά τροποποίηση να χρησιμοποιηθεί και ως χάρτης επιρροής. Το μειονέκτημα είναι η έλλειψη ακρίβειας σε περιπτώσεις όπου οι λεπτομέρειες είναι σημαντικές στη λήψη αποφάσεων.

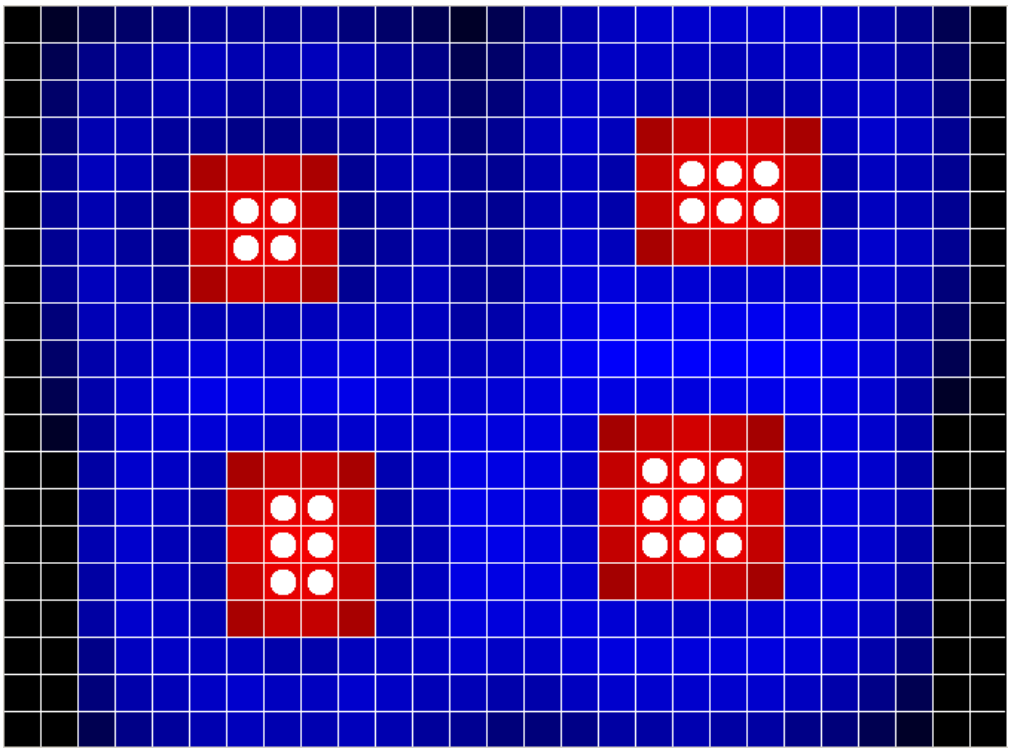


Ας δούμε μερικά παραδείγματα του είδους της πληροφορίας που μπορούμε να πάρουμε από έναν χάρτη επιρροής:

(1) Στο πεδίο της μάχης, μπορούμε πολύ εύκολα να αποφασίσουμε αν μια μονάδα βρίσκεται σε ευάλωτη θέση και πρέπει να οπισθοχωρήσει.



(2) Όταν θέλουμε να κατασκευάσουμε κάποιο κτήριο, μπορούμε πολύ εύκολα να βρούμε μια τοποθεσία κοντά στα υπόλοιπα κτήρια, αλλά όχι υπερβολικά κοντά (και προφανώς όχι επάνω σε αυτά).



3.2 Ευφυΐα σμήνους (swarm intelligence) [5]

Αυτή είναι μια τεχνική εμπνευσμένη από συστήματα που συναντάμε στη φύση και η οποία χαρακτηρίζεται από τις εξής ιδιότητες:

(1) Έχουμε ένα σύστημα στο οποίο πολλές μονάδες δουλεύουν αυτόνομα για την επίτευξη ενός κοινού στόχου. Λέγοντας «αυτόνομα» εννοούμε ότι δεν υπάρχει κεντρικός έλεγχος ο οποίος κατευθύνει τις μονάδες και ότι απαγορεύεται κάποια μονάδα να ελέγχεται άμεσα από κάποια άλλη.

(2) Οι μονάδες ακολουθούν ένα σύνολο από λίγες και απλές εντολές και έχουν περιορισμένο πεδίο αντίληψης. Δεν υπάρχει σαφές όριο στην ποσότητα της πληροφορίας στην οποία επιτρέπεται να έχουν πρόσβαση οι μονάδες, αλλά είναι καλό να διατηρείται μικρή ώστε η επεξεργασία της να γίνεται γρήγορα.

(3) Υπάρχει απλή, έμμεση επικοινωνία ανάμεσα στις μονάδες του συστήματος. Αυτό γίνεται συνήθως με δυο τρόπους. (α) Ο ένας τρόπος είναι μέσω του περιβάλλοντος στο οποίο «ζουν» οι μονάδες. Μια μονάδα μπορεί να προκαλέσει αλλαγές στο γειτονικό περιβάλλον της, τις οποίες οι άλλες μονάδες παρατηρούν και τροποποιούν τη συμπεριφορά τους. Αυτή η μέθοδος ονομάζεται «stigmetry». (β) Ο άλλος τρόπος με τον οποίο μια μονάδα μπορεί να επικοινωνήσει με τις υπόλοιπες είναι να αλλάξει την κατάστασή της (πχ θέση, ταχύτητα, κλπ…), ώστε όταν οι άλλες παρατηρήσουν αυτήν την αλλαγή να ανταποκριθούν ανάλογα.

Είναι αξιοσημείωτο ότι παρ' όλο που δεν υπάρχει κεντρικός έλεγχος του συστήματος που να κατευθύνει τις μονάδες, οι τοπικές αλληλεπιδράσεις ανάμεσα τους οδηγούν στην ανάδυση «έξυπνης» συμπεριφοράς, άγνωστης στις επί μέρους μονάδες.

Το μεγάλο πλεονέκτημα της τεχνικής αυτής είναι ο υψηλός λόγος αποτελεσματικότητας προς πολυπλοκότητα σχεδιασμού. Το μεγάλο μειονέκτημα είναι ότι προσφέρει προσεγγιστικές λύσεις και έτσι δεν είναι εφαρμόσιμη σε προβλήματα όπου απαιτούνται ακριβή αποτελέσματα. Επίσης, κατά το σχεδιασμό του συστήματος, είναι απαραίτητη η ρύθμιση των διαφόρων παραμέτρων ώστε να επιτευχθεί βέλτιστη απόδοση.

Στα videos micromanagement_1 και micromanagement_2 μπορούμε να δούμε πώς εφαρμόζεται αυτή η τεχνική στο πεδίο της μάχης. Στο πρώτο, έχουμε 2 marines εναντίον ενός zealot. Καθένας από τους δύο marines είναι προγραμματισμένος ως εξής:

-> Αν η απόσταση σου από τον εχθρό είναι μεγαλύτερη από 100 μονάδες, κάνε επίθεση στον εχθρό.
-> Αν η απόσταση σου από τον εχθρό είναι μικρότερη από 75 μονάδες, κάνε τα εξής:
  -> Για καθένα από τα οκτώ γειτονικά σου τετράγωνα, υπολόγισε την απόστασή του από το σύμμαχό σου, d1, και από τον εχθρό, d2.
  -> Βρες το τετράγωνο για το οποίο μεγιστοποιείται η παράσταση -1.75 * |d1 - 125| + 1.25 * d2 και πάνε προς αυτό.

Από αυτές τις απλές εντολές αναδύεται η εξής συμπεριφορά:

Όταν ο αντίπαλος zealot έχει σαν στόχο τον marine_1, αυτός κινείται σε κύκλο με κέντρο τον marine_2 και ακτίνα 125, ενώ ο marine_2 επιτίθεται στον zealot. Αν ο zealot αλλάξει στόχο, οι marine_1 και marine_2 αλλάζουν ρόλους.

Στο δεύτερο video, έχουμε την ίδια τεχνική, αλλά σε πιο μεγάλη κλίμακα. Έχουμε 15 marines εναντίον 12 zealots. Αρχικά, προσπάθησα να εφαρμόσω τον ίδιο αλγόριθμο με αυτόν του πρώτου video. Γρήγορα, όμως, συνάντησα ένα μεγάλο πρόβλημα: Όταν ένας marine είναι κοντά σε αντίπαλο zealot, πρέπει να λάβει υπόψη τις θέσεις όλων των γειτονικών marines και zealots για να αποφασίσει προς τα που θα κινηθεί. Αυτό σημαίνει ότι πρέπει να διατρέξω το σύνολο όλων των marines και zealots, να βρω αυτούς που είναι σχετικά κοντά στον τρέχον marine και μετά να χρησιμοποιήσω τις θέσεις όλων αυτών για να υπολογίσω το τετράγωνο στο οποίο πρέπει να κινηθώ. Στη χειρότερη περίπτωση, η πολυπλοκότητα του αλγορίθμου γίνεται O(N*(N+M)), όπου N το πλήθος των marines και M το πλήθος των zealots. Αποφάσισα, λοιπόν, να κάνω κάτι ελαφρώς διαφορετικό εδώ.

-> Αρχικά, φτιάχνω δύο χάρτες επιρροής, έναν για τους marines μου και έναν για τους zealots του αντιπάλου, ως εξής: Στις θέσεις γύρω από τους marines βάζω αρνητικούς αριθμούς οι οποίοι ελαττώνονται κατ' απόλυτη τιμή όσο απομακρυνόμαστε από τη θέση του marine, και προς το τέλος βάζω μικρούς θετικούς αριθμούς. Αυτό το κάνω γιατί θέλω οι marines να διατηρούν μια συγκεκριμένη απόσταση μεταξύ τους, και όχι απλώς να απομακρύνονται ο ένας από τον άλλον. Στις θέσεις γύρω από τους zealots βάζω απλώς αρνητικούς αριθμούς που ελαττώνονται κατ' απόλυτη τιμή όσο απομακρυνόμαστε από τη θέση του εκάστοτε zealot.

-> Έπειτα, για κάθε marine, κάνω τα εξής:
  -> Αν στο τετράγωνο στο οποίο βρίσκεται δεν υπάρχει επιρροή από εχθρικές μονάδες, κάνουμε επίθεση.
  -> Αν στο τετράγωνο στο οποίο βρίσκεται υπάρχει επιρροή από εχθρικές μονάδες, βρίσκουμε το βέλτιστο από τα οκτώ γειτονικά τετράγωνα, σύμφωνα με τους χάρτες επιρροής που έχουμε ετοιμάσει, και κινούμαστε σε αυτό.

Εύκολα προκύπτει ότι η πολυπλοκότητα στη χειρότερη περίπτωση είναι O(N + M). Βέβαια, υπάρχει κάποιο κόστος σε μνήμη (οι χάρτες επιρροής), αλλά αυτό είναι αμελητέο.

3.3 Finite state machines [1, 7, 8]

Για να μιλήσουμε για finite state machines, θα πρέπει να θυμηθούμε τι είναι ένα αυτόματο. Ένα αυτόματο, λοιπόν, είναι μια πεντάδα (Q, Σ, δ, q0, T) όπου…

-> Το Q είναι ένα πεπερασμένο σύνολο καταστάσεων.
-> Το Σ είναι ένα πεπερασμένο σύνολο συμβόλων
και καλείται αλφάβητο του αυτομάτου.
-> Η δ είναι μια συνάρτηση μετάβασης. Είναι δ : Q x Σ ->Q
-> Η q0 είναι στοιχείου του Q και ονομάζεται αρχική κατάσταση. Είναι η κατάσταση στην οποία βρίσκεται το αυτόματο πριν την επεξεργασία εισόδου.
-> Το T είναι ένα υποσύνολο του Q και καλείται σύνολο τελικών καταστάσεων.

Αν επεκτείνουμε το παραπάνω με ένα αλφάβητο εξόδου Γ και μια συνάρτηση εξόδου ω έχουμε έναν finite state transducer. Η συνάρτηση εξόδου μπορεί να εξαρτάται από την τρέχουσα κατάσταση και την τρέχουσα είσοδο (ω : Q x Σ -> Γ), οπότε λέμε ότι έχουμε μια μηχανή Mealy ή μπορεί να εξαρτάται μόνο από την τρέχουσα κατάσταση (ω : Q -> Γ), οπότε λέμε ότι έχουμε μια μηχανή Moore.

Η έννοια της finite state machine σε προγραμματιστικό context είναι λιγότερο αυστηρή από μαθηματική άποψη. Ορίζεται απλά ως ένα μοντέλο περιγραφής της συμπεριφοράς ενός συστήματος που χαρακτηρίζεται από τα εξής στοιχεία:

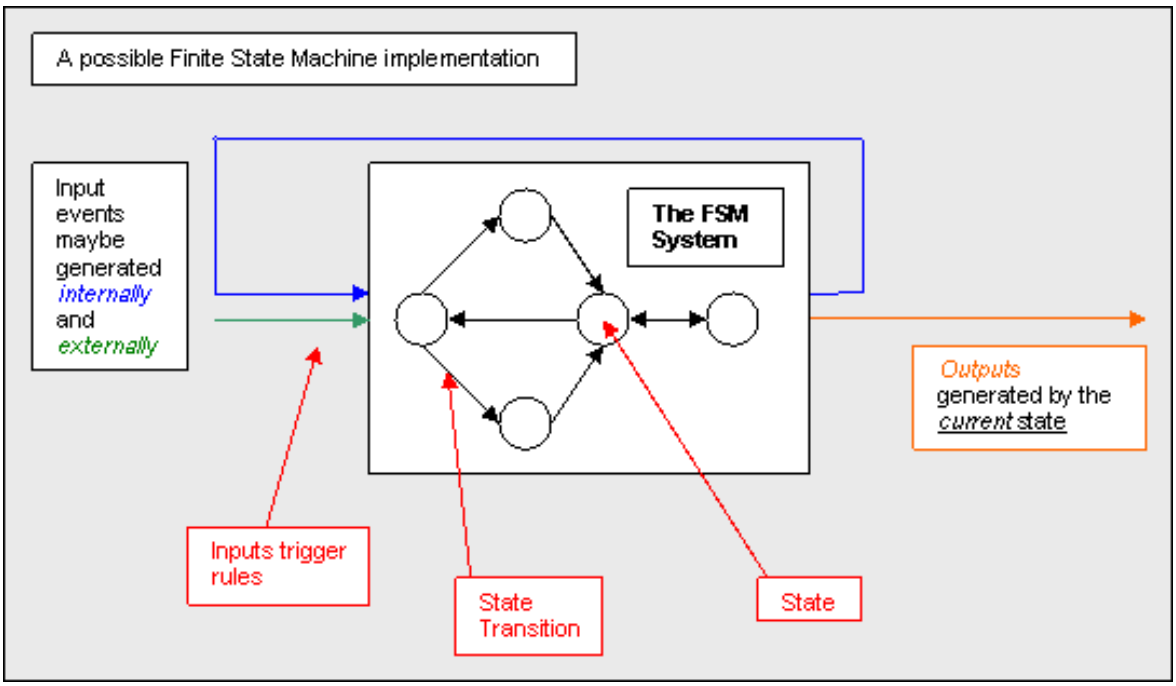
(1) Τις καταστάσεις στις οποίες μπορεί να βρίσκεται
το σύστημα (πεπερασμένες σε πλήθος).

(2) Μεταβάσεις από μια κατάσταση σε μια άλλη.

(3) Συνθήκες οι οποίες πρέπει να πληρούνται
για να επιτραπεί μια αλλαγή κατάστασης.

(4) Γεγονότα εισόδου, τα οποία δημιουργούνται είτε από
εξωτερικούς παράγοντες είτε από το ίδιο το σύστημα
και ενδέχεται να οδηγήσουν σε αλλαγή κατάστασης.

(5) Έξοδο, η οποία παράγεται από το σύστημα
και εξαρτάται από την τρέχουσα κατάσταση.

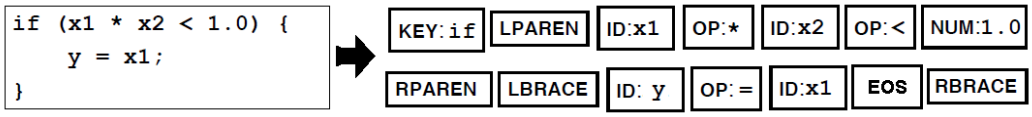


Τα πλεονεκτήματα της χρήσης finite state machines είναι ότι…

(1) Είναι πάρα πολύ εύκολες στο σχεδιασμό και την υλοποίηση.

(2) Υπάρχει άμεση αντιστοιχία στη συμπεριφορά του συστήματος και στον κώδικα όπου υλοποιείται αυτή η συμπεριφορά, γεγονός που οδηγεί σε κώδικα που είναι εύκολο να αποσφαλματωθεί, να συντηρηθεί και να επεκταθεί.

Ας δούμε ένα παράδειγμα από τον πραγματικό κόσμο. Ας υποθέσουμε ότι ξυπνάτε μια μέρα και αποφασίζετε να γράψετε έναν compiler (όχι, δεν είναι ακραίο παράδειγμα – γνωρίζω ανθρώπους που το κάνουν αυτό). Ένα από τα αρχικά στάδια της μεταγλώττισης ενός προγράμματος λέγεται lexical analysis. Σ' αυτό το στάδιο, το πρόγραμμα μετατρέπεται από μια ακολουθία χαρακτήρων σε μια ακολουθία από tokens (ένα token είναι μια ακολουθία χαρακτήρων που μαζί έχουν κάποιο συγκεκριμένο νόημα). Πολύ συχνά, αυτή η διαδικασία υλοποιείται ως finite state machine.



Εδώ, χρησιμοποιώ finite state machines σχεδόν παντού. Σχεδόν κάθε κομμάτι κώδικα έχει ένα μικρό μέρος το οποίο υλοποιείται ως finite state machine.

3.4 A* [6]

Ο A * (προφέρεται «έι σταρ») είναι ένας αλγόριθμος αναζήτησης που χρησιμοποιείται πάρα πολύ στη βιομηχανία παιχνιδιών, κυρίως στο πρόβλημα του pathfinding (εύρεση βέλτιστης (από άποψη μήκους) διαδρομής από το σημείο A στο σημείο B).

Προετοιμασία του αλγορίθμου

Για λόγους απλότητας θα υποθέσουμε ότι δουλεύουμε σε ένα δισδιάστατο πλέγμα (όπου κάθε τετράγωνο επικοινωνεί με τα οκτώ γειτονικά του και μόνο με αυτά). Θα χρειαστούμε δύο λίστες τετραγώνων, στις οποίες στο εξής θα αναφερόμαστε ως «ανοικτή λίστα» και «κλειστή λίστα». Σε κάθε τετράγωνο του πλέγματος αντιστοιχούμε τρεις αριθμούς, F, GκαιH, οι οποίοι αντιπροσωπεύουν…

-> Ο G το κόστος κίνησης από το αρχικό τετράγωνο έως το τρέχον τετράγωνο, ακολουθώντας το μονοπάτι που έχει δημιουργηθεί μέχρι την τρέχουσα χρονική στιγμή.

-> Ο H το εκτιμώμενο κόστος από το τρέχον τετράγωνο στο τετράγωνο-στόχο. Είναι σημαντικό ο H να μην υπερεκτιμά το κόστος από το τρέχον τετράγωνο στο τετράγωνο-στόχο, αν θέλουμε να βρούμε ένα βέλτιστο μονοπάτι.

-> Ο F το άθροισμα των δυο παραπάνω.

Επίσης, σε κάθε τετράγωνο αντιστοιχούμε ένα τετράγωνο-γονέα ώστε στο τέλος του αλγορίθμου να μπορούμε να βρούμε το μονοπάτι που παράγεται.

Εκτέλεση του αλγορίθμου

(1) Προσθέτουμε το αρχικό τετράγωνο στην ανοιχτή λίστα.

(2) Επαναλαμβάνουμε τα παρακάτω:

(α) Ψάχνουμε το τετράγωνο με το χαμηλότερο κόστος F στην ανοιχτή λίστα. Στο εξής θα αναφερόμαστε σε αυτό ως τετράγωνο-Α.

(β) Βγάζουμε το τετράγωνο-Α από την ανοιχτή λίστα και το βάζουμε στην κλειστή λίστα.

(γ) Για κάθε ένα από τα οκτώ γειτονικά τετράγωνα του τετραγώνου-Α

Αν δεν είναι walkable ή αν είναι στην κλειστή λίστα, το αγνοούμε.

Αλλιώς… Αν δεν είναι στην ανοιχτή λίστα, το βάζουμε. Κάνουμε το τετράγωνο-Α γονέα αυτού του τετραγώνου. Καταγράφουμε τα κόστη G, Η και F γι' αυτό το τετράγωνο.

Αν είναι ήδη στην ανοιχτή λίστα, ελέγχουμε αν το τρέχον μονοπάτι προς αυτό το τετράγωνο είναι καλύτερο, χρησιμοποιώντας το κόστος G σαν μέτρο σύγκρισης. Χαμηλότερο κόστος G σημαίνει καλύτερο μονοπάτι. Σε αυτήν την περίπτωση, αλλάζουμε τον γονέα του τετραγώνου στο τετράγωνο-Α και υπολογίζουμε ξανά τα κόστη G και F.

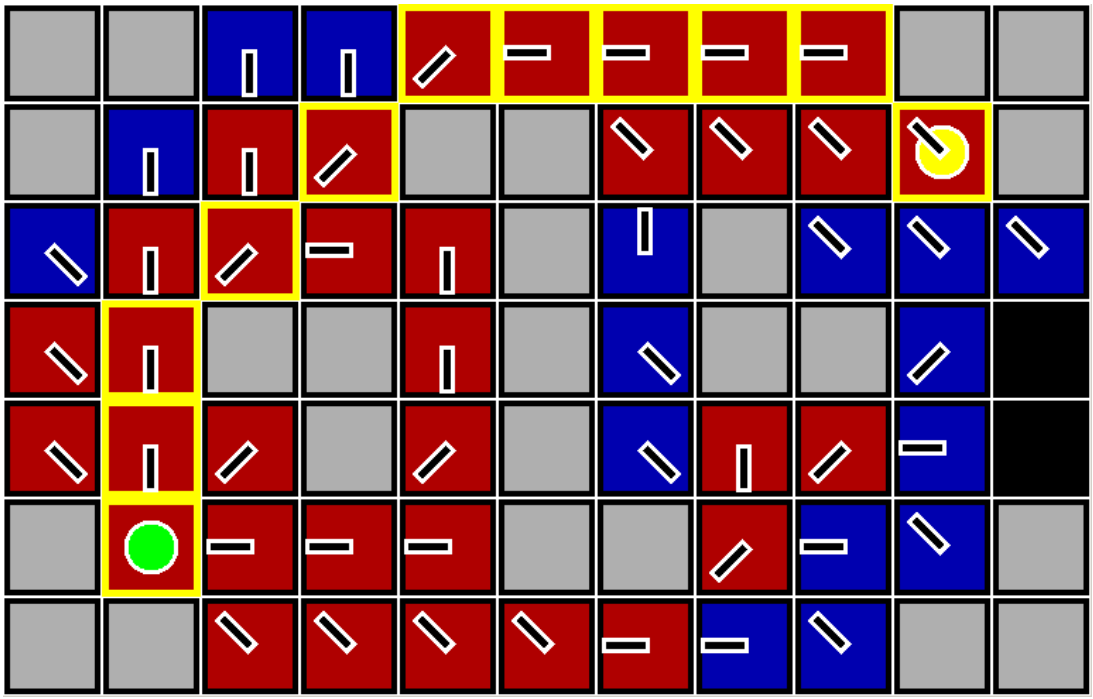
(δ) Σταματάμε όταν:

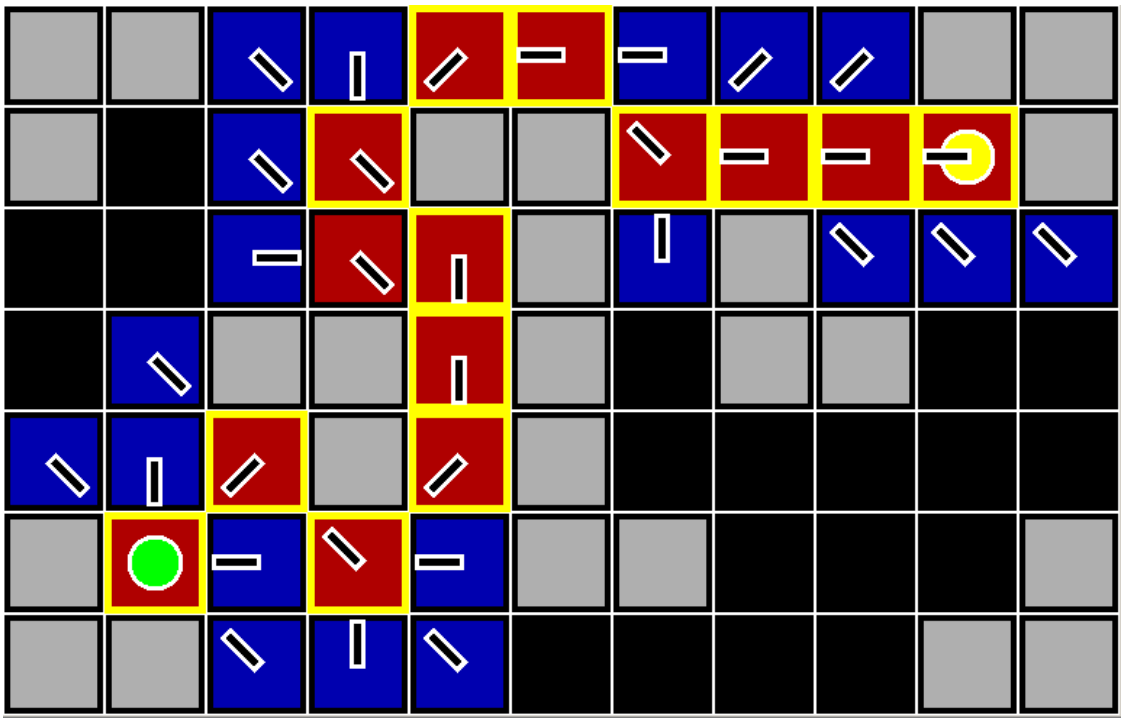
Προσθέτουμε το τετράγωνο-στόχο στην κλειστή λίστα
(σε αυτήν την περίπτωση υπάρχει μονοπάτι).

Αποτυγχάνουμε να βρούμε το τετράγωνο-στόχο και η ανοικτή λίστα είναι άδεια (σε αυτήν την περίπτωση δεν υπάρχει μονοπάτι).

3) Αν υπάρχει μονοπάτι, το βρίσκουμε πηγαίνοντας από το τετράγωνο στόχο πίσω στο αρχικό τετράγωνο μέσω των γονέων των τετραγώνων.

Ξέρω ότι δεν καταλάβατε και πολλά από αυτό, γι' αυτό έγραψα ένα μικρό πρόγραμμα ώστε να μπορούμε να τα αντιληφθούμε όλα αυτά με οπτικό τρόπο καθώς τρέχει ο αλγόριθμος. Παρατηρήστε ότι τη δεύτερη φορά χρησιμοποιώ μια συνάρτηση που υπερεκτιμά το κόστος Η και βρίσκω ένα μονοπάτι πολύ γρηγορότερα. Όμως το μονοπάτι που βρίσκω δεν είναι βέλτιστο.





Ανέφερα στην αρχή ότι ο A * χρησιμοποιείται πάρα πολύ στην βιομηχανία των παιχνιδιών. Η αλήθεια είναι ότι και το StarCraft χρησιμοποιεί τον A * για pathfinding. Γιατί, λοιπόν, τον παρουσιάζω εδώ; Τον παρουσιάζω γιατί τον χρησιμοποιώ με λίγο διαφορετικό τρόπο. Είναι πολύ εύκολο, τροποποιώντας κατάλληλα κάποιες από τις παραμέτρους του αλγορίθμου, να βρούμε όχι απλά το ελάχιστο σε μήκος μονοπάτι, αλλά και αυτό με το οποίο αποφεύγουμε κάποια στατικά εμπόδια που υπάρχουν ανάμεσα σ' εμάς και το στόχο.

Αξίζει να αναφέρουμε εδώ ότι υπάρχουν παραλλαγές του αλγορίθμου που μπορούν να βρουν το μονοπάτι με το ελάχιστο κόστος…

(i) από ένα αρχικό σε ένα από πολλά τελικά σημεία (B*)

(ii) από ένα αρχικό σε ένα τελικό σημείο με τις θέσεις των εμποδίων να μην είναι γνωστές από την αρχή, αλλά να εμφανίζονται κατά τη διάρκεια της εκτέλεσης του αλγορίθμου (D*)

3.5 Γενετικοί αλγόριθμοι (genetic algorithms) [9]

Οι γενετικοί αλγόριθμοι είναι μη ντετερμινιστικοί μέθοδοι αναζήτησης οι οποίες μιμούνται τη διαδικασία της φυσικής εξέλιξης. Χρησιμοποιούνται συνήθως όταν ο χώρος λύσεων του προβλήματος είναι μεγάλος, δεν υπάρχει αποτελεσματικός ντετερμινιστικός αλγόριθμος για την εύρεση καλής λύσης και μας ενδιαφέρει απλώς μια πολύ καλή λύση και όχι απαραίτητα η βέλτιστη. Η μεθοδολογία που ακολουθείται είναι η εξής:

(1) Παράγονται με τυχαίο τρόπο μερικές λύσεις και προστίθενται στον πληθυσμό από λύσεις του προβλήματος, ο οποίος αρχικά είναι άδειος.

(2) Σε κάθε λύση αντιστοιχούμε έναν αριθμό ο οποίος είναι ενδεικτικός της αποτελεσματικότητάς της.

(3) Διαλέγουμε μερικές καλές λύσεις και τις διασταυρώνουμε μεταξύ τους, με σκοπό τη δημιουργία νέων, καλύτερων λύσεων.

(4) Διαλέγουμε κάποιες λύσεις και τις μεταλλάσσουμε (προκαλούμε μικρές αλλαγές), με σκοπό τη δημιουργία νέων, καλύτερων λύσεων.

(5) Επαναλαμβάνουμε τα βήματα (1) έως (4) αρκετές φορές, και κρατάμε στο τέλος την καλύτερη από τις λύσεις που υπάρχουν στον πληθυσμό.

Εδώ χρησιμοποιώ γενετικούς αλγορίθμους σε συνδυασμό με χάρτες επιρροής για να βρω κατάλληλη θέση όποτε θέλω να κατασκευάσω ένα κτήριο.

4. Παρουσίαση του κώδικα

Όταν ξεκίνησα να φτιάχνω το πρόγραμμα έκανα ένα σχεδιάγραμμα σαν κι αυτό:

main goal:
-> win the game

sub goals:
-> destroy enemy buildings

main goal:
-> destroy enemy buildings

sub goals:
-> build a strong army
-> use army to attack the enemy

main goal:
-> build a strong army

sub goals:
-> gather resources
-> build factories
-> train soldiers
-> make upgrades

main goal:
->use army to attack the enemy

sub goals:
-> scout the map to locate enemy
-> control attack squads during battle

main goal:
-> gather resources

sub goals:
-> train workers
-> put workers to work

main goal:
-> build factories

sub goals:
-> find a suitable location
-> order a worker to build

κλπ...

Βλέπετε ότι γίνεται πολύ περίπλοκο πολύ γρήγορα. Για να αντιμετωπίσω αυτήν την πολυπλοκότητα αποφάσισα να οργανώσω τον κώδικα σε κλάσεις (στη C++, πολύ χοντρικά, μια 'κλάση' είναι ένα μια συλλογή δεδομένων (μεταβλητών) και διαδικασιών (συναρτήσεων) που έχει σαν σκοπό την επίλυση ενός συγκεκριμένου προβλήματος) ως εξής…

4.1 Γενική περιγραφή του κώδικα

Η κλάση DocBot

Αυτή είναι η κύρια κλάση του προγράμματος. Σ' αυτήν μπορώ να περιγράψω τη στρατηγική που θέλω να ακολουθήσω σε σχετικά υψηλό επίπεδο.

Έπειτα, αυτές οι οδηγίες περνούν σε μία ή περισσότερες από τις παρακάτω κλάσεις, οι οποίες φροντίζουν να τις μετατρέψουν σε εντολές χαμηλού επιπέδου, τις οποίες και αντιλαμβάνεται το παιχνίδι.

Ανέφερα στην αρχή ότι το bot είναι εύκολα επεκτάσιμο με νέες στρατηγικές. Αυτός είναι ένας από τους λόγους για τους οποίους συμβαίνει αυτό -> το γεγονός ότι έχω διαχωρίσει το κομμάτι του κώδικα στο οποίο περιγράφω τη στρατηγική που θα ακολουθήσω από το υπόλοιπο πρόγραμμα.

Η κλάση BuildManager

Αυτή η κλάση είναι υπεύθυνη για την κατασκευή κτηρίων και την εκπαίδευση μονάδων (εργατών και στρατιωτών). Περιέχει μια λίστα από εντολές παραγωγής στην οποία μπορούν να προσθέτουν οι υπόλοιπες κλάσεις του προγράμματος όποτε χρειάζεται.

Κάθε εντολή παραγωγής περιέχει πληροφορίες για το είδος της μονάδας που παράγεται, το σκοπό για τον οποίο παράγεται, την τρέχουσα κατάσταση της παραγωγής (Zero, Awaiting_Init, In_Progress, Completed, Error) κλπ... και σε κάθε επανάληψη του κύριου βρόχου του προγράμματος διατρέχεται όλη η λίστα και λαμβάνονται αποφάσεις με βάση τα events του παιχνιδιού και την τρέχουσα κατάσταση της παραγωγής.

Να σημειώσουμε εδώ ότι ο BuildManager είναι υπεύθυνος και για αυτό που λέμε building dependency resolution. Αυτό σημαίνει ότι αν εγώ του ζητήσω να εκπαιδεύσει μια μονάδα Α η οποία χρειάζεται το κτήριο Β, ο BuildManager θα βάλει μόνος του το κτήριο Β πριν τη μονάδα Α στη λίστα. Επίσης αν αυτό το κτήριο Β χρειάζεται ένα κτήριο Γ, αυτό θα μπει αυτόματα πριν από το Β στη λίστα παραγωγής κλπ...

Η κλάση BuildPosPlanner

Όταν ο BuildManager θέλει να χτίσει ένα κτήριο, συμβουλεύεται αυτήν την κλάση για να βρει κατάλληλη τοποθεσία. Σε πρώτη φάση φτιάχνουμε έναν πίνακα μ χ ν (ο οποίος αντιστοιχεί στο χάρτη του παιχνιδιού) και σε κάθε θέση του οποίου βάζουμε μία τιμή που αντιπροσωπεύει την επιθυμία μας να χτίσουμε στο αντίστοιχο κουτάκι του χάρτη του παιχνιδιού. Στις θέσεις όπου υπάρχουν ήδη κτήρια και στις θέσεις που δεν μπορούμε να χτίσουμε λόγω ακατάλληλου εδάφους βάζουμε μεγάλους αρνητικούς

αριθμούς. Στις θέσεις κοντά στην αρχική μας βάση και στις θέσεις γύρω από τα κτήριά μας βάζουμε θετικούς αριθμούς (influence mapping).

Έπειτα, με βάση αυτήν την πληροφορία εφαρμόζω έναν γενετικό αλγόριθμο για να βρω μια καλή θέση για το κτήριό μου. Ξεκινάω με πολλές τυχαίες θέσεις και υπολογίζω για κάθε μία πόσο καλή είναι. Αυτές που είναι πολύ καλές τις κρατάω. Κάποιες από αυτές τις μεταλλάσσω και τις διασταυρώνω μεταξύ τους με σκοπό να μου προκύψουν άλλες, καλύτερες. [Πχ αν μεταλλάξω τη θέση (10, 20) θα πάρω μια από τις (9, 20), (11, 20), (10, 19), (10, 21), και αν διασταυρώσω τις (10, 20) και (15, 30) θα πάρω μία από τις (10, 30), (15, 20), (12.5, 20), (12.5, 30), (10, 25), (15, 25), (12.5, 25) κλπ...] Επαναλαμβάνω αυτήν τη διαδικασία αρκετές φορές και στο τέλος κρατάω την καλύτερη λύση.

Η κλάση MiningManager

Αυτή η κλάση είναι υπεύθυνη για τη διαχείριση των υλικών πόρων. Στην αρχή του παιχνιδιού και όποια άλλη στιγμή χρειαστεί παραγγέλνει εργάτες από τον BuildManager και τους βάζει για δουλειά.

Οι κλάσεις AttackManager & ReconManager

Αυτές οι δύο κλάσεις έχουν παρόμοια δομή. Δέχονται εντολές από την DocBot για να εκπαιδεύσουν στρατιώτες, μεταβιβάζουν αυτές τις εντολές στην BuildManager και ειδοποιούν όποτε η παραγγελία είναι έτοιμη. Η διαφορά είναι ότι η πρώτη έχει σκοπό να χτίσει στρατιώτες για επίθεση ενώ η δεύτερη για λόγους εξερεύνησης του χάρτη (πχ για να βρούμε την τοποθεσία του εχθρού).

Οι κλάσεις AttackSquad & ReconSquad

Αυτές οι δύο κλάσεις αντιπροσωπεύουν ομάδες επίθεσης και ανίχνευσης του χάρτη, αντίστοιχα. Η DocBot έχει πρόσβαση σ' αυτές μέσω των AttackManager & ReconManager αντίστοιχα.

4.2 Usecases: Τι συμβαίνει όταν…

…θέλω να εκπαιδεύσω έναν εργάτη για συλλογή πρώτων υλών;

Η εντολή που θα χρησιμοποιήσω στο σημείο
όπου περιγράφω τη στρατηγική μου είναι η εξής:

```
mining_manager.OrderWorkers(1, true);
```

… αν θέλω τον εργάτη για minerals, και

```
mining_manager.OrderWorkers(1, false);
```

… αν τον θέλω για vespene gas.

Η συνάρτηση αυτή με τη σειρά της προωθεί
την παραγγελία στον build_manager ως εξής:

```
//...

(*(bot->build_manager))  << BuildManager::BuildOrderInfo(
        BWAPI::UnitTypes::Terran_SCV,
        BuildManager::BuildOrderInfo::Purpose::MINE_MINERALS);

//...
```

και

```
//...

(*(bot->build_manager))  << BuildManager::BuildOrderInfo(
        BWAPI::UnitTypes::Terran_SCV,
        BuildManager::BuildOrderInfo::Purpose::MINE_GAS);

//...
```

αντίστοιχα.

Ο build_manager παραλαμβάνει την παραγγελία
και την προσθέτει στη λίστα με τις υπόλοιπες:

```
BuildManager& operator << (const BuildOrderInfo & build_order_info)
{
   return build_order.push_back(build_order_info), *this;
}
```

Πλέον, η συνάρτηση Step του BuildManager που καλείται σε κάθε επανάληψη του κύριου βρόχου του προγράμματος, αναλαμβάνει την εκτέλεση της παραγγελίας. Αρχικά, η κατάσταση της παραγγελίας είναι ορισμένη ως Init. Η παραγγελία εξετάζεται μόνο αν είναι η πρώτη στη λίστα ή αν όλες οι προηγούμενες βρίσκονται είτε στην κατάσταση In_Progress είτε στην κατάσταση Completed. Αρχικά ελέγχεται αν υπάρχουν οι απαραίτητες προϋποθέσεις για να εκτελεστεί η παραγγελία. Αν δεν υπάρχει κάποιο κτήριο που είναι απαραίτητο, εισάγουμε μια παραγγελία γι' αυτό το κτήριο ακριβώς πριν από την τρέχουσα. Αν δεν υπάρχουν αρκετές πρώτες ύλες, περιμένουμε.

Αν όλα είναι εντάξει, βρίσκουμε ένα κατάλληλο κτήριο και δίνουμε εντολή για εκπαίδευση του εργάτη και αλλάζουμε την κατάσταση της παραγγελίας σε Awaiting_Init. Όταν το κτήριο αρχίσει την εκπαίδευση, λαμβάνουμε ένα UnitCreated event από το παιχνίδι και αλλάζουμε την κατάσταση σε In_Progress. Όταν τελειώσουμε, αλλάζουμε την κατάσταση σε Completed. Έπειτα, επιστρέφουμε τον εργάτη στον manager που τον ζήτησε (στον mining_manager σε αυτήν την περίπτωση).

```
void BuildManager::Step(const std::list<BWAPI::Event> & event_list)
{

//...

  if (cur_it->status == BuildStatus::Completed)
  {
    //...
```

```
   switch (cur_it->purpose)
   {
      //...

      case BuildOrderInfo::Purpose::MINE_MINERALS:
      bot->mining_manager->AddToWorkforce(cur_it->creation, true); break;

      case BuildOrderInfo::Purpose::MINE_GAS:
      bot->mining_manager->AddToWorkforce(cur_it->creation, false); break;

      //...
   }
 }

//...
}

void MiningManager::AddToWorkforce(BWAPI::Unit * unit, bool for_minerals = true)
{
   if (for_minerals) mineral_miners.insert(unit);
   else gas_miners.insert(unit);
}
```

Πλέον, ο εργάτης είναι έτοιμος για δουλειά. Αυτό το αναλαμβάνει η συνάρτηση Step του mining_manager.

<u>… θέλω να εκπαιδεύσω έναν εργάτη για κατασκευή κτηρίων;</u>

Η εντολή που θα χρησιμοποιήσω στη DocBot είναι η εξής:

```
build_manager.OrderWorkers(1);
```

Ο build_manager αναλαμβάνει να προσθέσει την παραγγελία στη λίστα…

```
void OrderWorkers(int count)
{
```

```
  //...

  (*this) <<BuildManager::BuildOrderInfo(
          BWAPI::UnitTypes::Terran_SCV,
          BuildManager::BuildOrderInfo::Purpose::BUILDING, 0, 0);

//...
}
```

Η διαδικασία που ακολουθείται από εδώ και πέρα είναι ίδια με την παραπάνω, εκτός από το τελευταίο στάδιο. Όταν η κατάσταση της παραγγελίας γίνει Completed, ο εργάτης πηγαίνει στη λίστα με τους εργάτες του build_manager...

```
void BuildManager::Step(const std::list<BWAPI::Event> & event_list)
{

//...

  if (cur_it->status == BuildStatus::Completed)
  {
    //...

    switch (cur_it->purpose)
    {
        //...

        case BuildOrderInfo::Purpose::BUILDING:
        bot->build_manager->AddToWorkforce(cur_it->creation); break;

        //...
    }
  }

//...
}
```

…θέλω να κατασκευάσω ένα κτήριο;

Ας υποθέσουμε ότι θέλω να χτίσω ένα Barracks (παράγει στρατιώτες) και ένα Supply Depot (αυξάνει το όριο των μονάδων που μπορώ να παράγω).

Οι εντολές που χρησιμοποιώ στη DocBot είναι:

```
build_manager<< BWAPI::UnitTypes::Terran_Barracks;
build_manager<< BWAPI::UnitTypes::Terran_Supply_Depot;
```

Κάθε μία απ' αυτές προσθέτει μια παραγγελία στη λίστα του build_manager...

```
BuildManager &  operator << (BWAPI::UnitType unit_type)
{
        build_order.push_back(BuildOrderInfo(unit_type,
        BuildOrderInfo::Purpose::NONE, 0, 0));

        return *this;
}
```

Στη συνάρτηση Step του build_manager τώρα, κάθε μια από τις παραγγελίες βρίσκεται στην κατάσταση Init. Αν πληρούνται οι προϋποθέσεις για την κατασκευή του κτηρίου (πχ υπάρχουν τα απαραίτητα κτήρια, υπάρχουν οι πρώτες ύλες κλπ…) και υπάρχει ελεύθερος εργάτης, βρίσκουμε μια κατάλληλη θέση για το κτήριο με τη βοήθεια του BuildPosPlanner, δίνουμε εντολή στον εργάτη να χτίσει σε εκείνη τη θέση και περνάμε στην κατάσταση Awaiting_Init. Όταν ο εργάτης ξεκινήσει το χτίσιμο περνάμε στην In_Progress και όταν τελειώσει στην Completed.

Ο BuildManager αναθέτει την εύρεση κατάλληλης θέσης στον BuildPosPlanner:

```
void BuildManager::Step(const std::list<BWAPI::Event> & event_list)
{
   //...

        build_pos_planner.building_type = creation_type;
```

```
        build_pos_planner.FindGoodPos(building_pos);

    //...
}

bool BuildPosPlanner::FindGoodPos(BWAPI::TilePosition & position)
{
    /* Πρώτα φτιάχνουμε ένα χάρτη επιρροής ο οποίος θα χρησιμοποιηθεί
       από τη fitness function του γενετικού μας αλγορίθμου. */

    BuildInfluenceMap();

    //...

    pool.clear();

    /* Έπειτα, τρέχουμε το γενετικό αλγόριθμο */

    while (true)
    {
        SpawnAndAdd(50);

        std::sort(pool.begin(), pool.end());
        pool.erase(std::unique(pool.begin(), pool.end()), pool.end());

        if (count++ == max_count) break;

        SpawnAndAdd(1);

        if (pool.size() > 100) pool.resize(50);

        for (unsigned i = 0; i<pool.size() / 3; ++i)
        {
            MutateAndAdd(pool[i]);
            CrossbreedAndAdd(pool[i], pool[i+1]);
        }
```

```
    }

  position = pool[0].position;

  return true;
}

void BuildPosPlanner::BuildInfluenceMap()
{
    const unsigned map_width = game->mapWidth();
    const unsigned map_height = game->mapHeight();

    const double big_negative_val = -1e4;

    /* Αρχικοποίηση του χάρτη επιρροής */

    influence_map.clear();
    influence_map.resize(map_width);

    for (unsigned i = 0; i<map_width; ++i)
          influence_map[i].resize(map_height, 0);

    Line magic_line;

    BWAPI::TilePosition command_center;
    BWAPI::TilePosition map_center;

    command_center = game->self()->getStartLocation();
    map_center = BWAPI::TilePosition(map_width/2, map_height/2);

    /* Η μαγική γραμμή είναι η νοητή γραμμή που ενώνει
       το κεντρικό μας κτήριο με το κέντρο του χάρτη */

    magic_line = GetLineFromPoints(command_center, map_center);

    double magic_line_coef = -25;
```

```
double magic_line_const = 25;
double command_center_coef = 40;
bool invalidate_surroundings = true;

/* Χτίζουμε τα Supply Depots μακριά από τη μαγική
   γραμμή και τα υπόλοιπα κτήρια κοντά σε αυτήν. */

if (building_type == BWAPI::UnitTypes::Terran_Supply_Depot)
{
   magic_line_coef = 50;
   magic_line_const = 0;
   invalidate_surroundings = false;
}

const double distance_normalizer =
   sqrt(double(map_width * map_width + map_height * map_height));

for (unsigned i = 0; i<map_width; ++i)
{
   for (unsigned j = 0; j <map_height; ++j)
   {
      /* Δεν μπορούμε να χτίσουμε αν δεν το επιτρέπει το έδαφος */

      if (!game->isBuildable(i, j)) influence_map[i][j] = big_negative_val;

      influence_map[i][j] +=
      magic_line_const +
      magic_line_coef *
      LinePointDistance(magic_line,
      BWAPI::TilePosition(i,j)) / distance_normalizer;

      influence_map[i][j] +=
      command_center_coef *
         (1 - PointPointDistance(command_center,
      BWAPI::TilePosition(i,j)) /distance_normalizer);
   }
```

```
    }

    std::set<BWAPI::Unit *> minerals = game->getStaticMinerals();
    std::set<BWAPI::Unit *> geysers = game->getStaticGeysers();

    for (std::set<BWAPI::Unit *>::iterator it = minerals.begin(); it != minerals.end(); ++it)
    {
        /* Δεν μπορούμε να χτίσουμε πάνω στα minerals, ούτε κοντά σε αυτά. */
    }

    for (std::set<BWAPI::Unit *>::iterator it = geysers.begin(); it != geysers.end(); ++it)
    {
        /* Δεν μπορούμε να χτίσουμε πάνω στους
          vespene geysers, ούτε κοντά σε αυτούς. */
    }

    std::set<BWAPI::Unit *>player_units = game->self()->getUnits();
    std::set<BWAPI::Unit *>player_buildings;

    for (std::set<BWAPI::Unit *>::iterator it = player_units.begin();
        it != player_units.end(); ++it)
            if ((*it)->getType().isBuilding()) player_buildings.insert(*it);

    for (std::set<BWAPI::Unit *>::iterator it = player_buildings.begin();
        it != player_buildings.end(); ++it)
        {
            /* Μπορούμε να χτίσουμε κοντά στα κτήρια μας αλλά όχι πάνω σε αυτά.
               Ακόμη, μπορούμε να χτίζουμε τα Supply Depots όσο κοντά σε άλλα
               Supply Depots θέλουμε, αλλά για τα υπόλοιπα κτήρια πρέπει να αφήνουμε
              περισσότερο ελεύθερο χώρο (για τις μονάδες που παράγονται από αυτά). */
        }
}

void BuildPosPlanner::Evaluate(Solution&solution)
{
    /* Υπολογίζουμε την αποτελεσματικότητα της λύσης (θέσης),
```

```
      αθροίζοντας τις αντίστοιχες τιμές του χάρτη επιρροής. */
}

void BuildPosPlanner::SpawnAndAdd(unsigned number)
{
   /* Δημιουργούμε number σε πλήθος τυχαίες λύσεις,
      υπολογίζουμε την αποτελεσματικότητα τους με την Evaluate
      και τις προσθέτουμε στη λίστα με τις υπόλοιπες λύσεις. */
}

void BuildPosPlanner::MutateAndAdd(const Solution & solution)
{
   /* Δημιουργούμε μια καινούρια λύση μεταλλάσσοντας μία υπάρχουσα.
      Η μετάλλαξη γίνεται ως εξής:
        -> με 40% πιθανότητα αλλάζουμε τη θέση της λύσης στον άξονα x.
        -> με 40% πιθανότητα αλλάζουμε τη θέση της λύσης στον άξονα y.
        -> με 20% πιθανότητα αλλάζουμε τη θέση της λύσης και στους δύο άξονες.
       Όποτε γίνεται αλλαγή, αυτή είναι είτε αύξηση είτε μείωση κατά μία
       μονάδα (κάθε περίπτωση έχει πιθανότητα 50% να πραγματοποιηθεί).
      Έπειτα, υπολογίζουμε την αποτελεσματικότητα της λύσης
      και την προσθέτουμε στη λίστα με τις υπόλοιπες. */
}

void BuildPosPlanner::CrossbreedAndAdd(const Solution & solution1,
          const Solution & solution2)
{
   /* Δημιουργούμε μια καινούρια λύση διασταυρώνοντας δύο υπάρχουσες ως εξής:
      Για θέση της καινούριας λύσης στον άξονα χ βάζουμε:
         -> με 33% πιθανότητα τη θέση της solution1 στον άξονα χ
         -> με 33% πιθανότητα τη θέση της solution2 στον άξονα χ
         -> με 33% πιθανότητα το ημιάθροισμα των
            θέσεων των δύο λύσεων στον άξονα χ
      Όμοια για τη θέση της καινούριας λύσης στον άξονα y.
      Υπολογίζουμε την αποτελεσματικότητα της καινούριας
      λύσης και την προσθέτουμε στη λίστα με τις υπόλοιπες. */
}
```

…θέλω να φτιάξω δυο ομάδες εξερεύνησης του χάρτη;

Γράφω στη DocBot τα εξής:

```
recon_manager.OrderSquad(0, BWAPI::UnitTypes::Terran_SCV,1);
recon_manager.OrderSquad(0, BWAPI::UnitTypes::Terran_SCV,2);

recon_manager.squads[1].target =
*recon_manager.possible_emeny_start_locations.begin();

recon_manager.squads[2].target =
*(recon_manager.possible_emeny_start_locations.rbegin());

recon_manager.squads[1].recon_state =
ReconSquad::ReconState::MOVE_TO_TARGET;
recon_manager.squads[2].recon_state =
ReconSquad::ReconState::MOVE_TO_TARGET;
```

Τα πρώτα δύο statements παραγγέλνουν από τον build_manager δυο εργάτες και όταν είναι έτοιμοι τους στέλνουν στον recon_manager, τον έναν στην ομάδα με id 1 και τον άλλον στην ομάδα με id2.

Τα επόμενα δύο δίνουν σε καθεμία από τις δυο ομάδες ένα στόχο προς εξερεύνηση και τα τελευταία δυο αλλάζουν την κατάστασή τους έτσι ώστε όταν είναι έτοιμες να ξεκινήσουν την εξερεύνηση.

Ας δούμε τώρα τι συμβαίνει στη συνάρτηση Step ενός ReconSquad…

```
void ReconSquad::Step(const std::list<BWAPI::Event> & event_list)
{
   if (build_state == BuildState::BEING_BUILT)
   {
      // Αν είναι έτοιμη η ομάδα, αλλάζουμε
     // την κατάσταση σε BuildState::READY.
   }
   else // build_state == BuildState::READY
```

```
  {
    for (std::list<BWAPI::Event>::const_iterator it = event_list.begin();
      it != event_list.end(); ++it)
    {
      /* Ελέγχουμε τα events που παίρνουμε απ' το παιχνίδι, και αν
      έχει βρεθεί το κεντρικό κτήριο του εχθρού ειδοποιούμε τον
      recon_manager. Αν αντιληφθούμε ότι μας κυνηγάνε,
      μένουμε εκεί (περνάμε στην ReconState::STAY_THERE) */
    }

    switch(recon_state)
    {
      case ReconState::IDLE: {} break;

      case ReconState::MOVE_TO_TARGET:
      {
        /* Πλησιάζουμε το στόχο μέχρι να φτάσουμε αρκετά κοντά.
        Αν όταν φτάσουμε δε βρούμε το κεντρικό κτήριο του αντιπάλου,
        αφαιρούμε αυτήν την τοποθεσία από τις πιθανές τοποθεσίες
        του εχθρού. Περνάμε στη ReconState::RUN_AWAY */
      } break;

      case ReconState::STAY_THERE: {} break;

      case ReconState::RUN_AWAY:
      {
        /* Επιστρέφουμε στη βάση μας. Όταν φτάσουμε
          περνάμε στη ReconState::IDLE. */
      }

    default: break;
    }
  }
}
```

…θέλω να φτιάξω μια ομάδα στρατιωτών για επίθεση;

```
typedef std::pair<BWAPI::UnitType, int> SquadOrder;

SquadOrder squad_orders[] =
{
   SquadOrder(BWAPI::UnitTypes::Terran_Marine, 2)
   SquadOrder(BWAPI::UnitTypes::Terran_Firebat, 2),
   SquadOrder(BWAPI::UnitTypes::Terran_Medic, 1)
};

attack_manager.OrderSquad(0, squad_orders, squad_orders + 3, 1);
attack_manager.OrderSquad(0, squad_orders, squad_orders + 3, 1);
attack_manager.OrderSquad(0, BWAPI::UnitTypes::Terran_Medic, 1);

attack_manager.squads[1].target = recon_manager.enemy_start_location;
attack_manager.squads[1].attack_state = AttackSquad::AttackState::CLUSTER;
```

Αυτή η γραμμή:

```
attack_manager.OrderSquad(0, squad_orders, squad_orders + 3, 1);
```

παραγγέλνει 2 marines, 2 firebats και ένα medic και τα βάζει στην ομάδα 1.

Αυτή η γραμμή:

```
attack_manager.OrderSquad(0, BWAPI::UnitTypes::Terran_Medic, 1);
```

παραγγέλνει ένα medic και το βάζει στην ομάδα 1.

Οι παραγγελίες εκτελούνται από τον build_manager με τρόπο παρόμοιο με αυτούς που περιγράψαμε παραπάνω.

Οι τελευταίες δυο γραμμές θέτουν το στόχο επίθεσης της ομάδας 1 και αλλάζουν την κατάστασή της ώστε όταν ολοκληρωθεί η εκπαίδευση των μονάδων να ξεκινήσει η επίθεση.

Ας ρίξουμε μια ματιά στην AttackSquad::Step…

```
void AttackSquad::Step(const std::list<BWAPI::Event> & event_list)
{
   if (disabled) return; // Αυτό θα δείτε αργότερα γιατί είναι απαραίτητο.

   BWAPI::TilePosition center(0,0);

   int alive_count = 0;

   for (std::set<BWAPI::Unit *>::iterator it = units.begin(); it != units.end(); ++it)
   {
      if (!(*it)->exists()) continue;

      alive_count++;

      center += (*it)->getTilePosition();
   }

   if (alive_count == 0) return;

   center = BWAPI::TilePosition(center.x() / alive_count, center.y() / alive_count);

   for (std::list<BWAPI::Event>::const_iterator event_it = event_list.begin();
      event_it != event_list.end(); ++event_it)
   {
      /* Ελέγχουμε τα events του παιχνιδιού και ενημερώνουμε με βάση
         αυτά τα σύνολα των δικών μας και των εχθρικών μονάδων. */
   }

   if (build_state == BuildState::BEING_BUILT)
   {
         /* Αν δεν είναι έτοιμη η ομάδα, δεν κάνουμε τίποτα.
         Αν είναι έτοιμη, περνάμε στην BuildState::READY.*/
   }
```

```
    else // build_state == BuildState::Ready
    {

        /* Κάνουμε κάποιες αλλαγές κατάστασης ανάλογα με το
        αν υπάρχουν εχθροί κοντά στην ομάδα και ανάλογα με το
        χρόνο για τον οποίο βρισκόμαστε στην τρέχουσα κατάσταση. */

      switch(attack_state)
      {
         case AttackState::IDLE: {} break;
         case AttackState::CLUSTER:
         { /* Συσπειρώνουμε τις μονάδες της ομάδας. */ } break;

         case AttackState::MOVE_TO_TARGET:
         {
              /* Κινούμαστε προς το στόχο. Αν είμαστε αρκετά κοντά,
                 περνάμε στην AttackState::FIRE_AT_WILL. */
         } break;

        case AttackState::ATTACK:
        {
         /* Αν δεν έχουμε στόχο, επιλέγουμε κατάλληλα έναν από το σύνολο
            των εχθρικών μονάδων. Αν είμαστε μακριά από το στόχο, κινούμαστε
            προς αυτόν. Αν είμαστε κοντά, κάνουμε επίθεση. */
        } break;

      case AttackState::FIRE_AT_WILL: {} break;

      default: break;
     }
   }
}
```

…θέλω να κάνω επίθεση με tanks;

Υποθέτοντας ότι έχουμε έτοιμες δυο ομάδες μονάδων, μία με στρατιώτες και μία με tanks, κάνουμε τα εξής:

```
attack_manager.squads[1].disabled = true;
attack_manager.squads[2].disabled = true;

siege_attack = new SiegeAttack(attack_manager.squads[2].units,
attack_manager.squads[1].units, recon_manager.enemy_start_location);
```

Οι δυο πρώτες γραμμές είναι απαραίτητες, ώστε οι ομάδες να μην ελέγχονται από την AttackSquad::Step.

Ας ρίξουμε μια ματιά στην SiegeAttack::Step…

```
void Step(const std::list<BWAPI::Event>& events)
{
   for (std::list<BWAPI::Event>::const_iterator event_it = events.begin();
      event_it != events.end(); ++event_it)
   {
      /* Ελέγχουμε τα events του παιχνιδιού και ανανεώνουμε κατάλληλα
         τα σύνολα των δικών μας και των εχθρικών μονάδων */
    }

   /* Κάνουμε κάποιες αλλαγές κατάστασης ανάλογα με το αν
   υπάρχουν εχθροί κοντά στην ομάδα και ανάλογα με το χρόνο
   για τον οποίο βρισκόμαστε στην τρέχουσα κατάσταση. */

   if (state == State::SIEGE1)
   {
      /* Αν είμαστε πολύ κοντά στα αντίπαλα κτήρια απομακρυνόμαστε.
          Αν είμαστε πολύ μακριά από τα αντίπαλα κτήρια, πλησιάζουμε.
          Αν είμαστε στην κατάλληλη απόσταση, περνάμε στη State::SIEGE2 */
   }
```

```
    if (state == State::SIEGE2)
    {
       /* Αν υπάρχουν εχθρικά κτήρια δίπλα, κάνουμε επίθεση,
          αλλιώς περνάμε στην State::CLUSTER2 */
    }

    if (state == State::ATTACK)
    {
       /* Διαλέγουμε μια από τις εχθρικές μονάδες που
          βρίσκονται κοντά και κάνουμε επίθεση.*/
    }

    if (state == State::CLUSTER2)
    {
       /* Κινούμε όλες τις μονάδες προς το κέντρο των tanks. */
    }

    if (state == State::CLUSTER1)
    {
        /* Απομακρύνουμε τους στρατιώτες τον έναν από τον άλλον
           και κινούμε τα tanks προς το κέντρο των στρατιωτών. */
    }

    if (state == State::MOVE)
    {
       /* Κινούμε τις μονάδες προς το στόχο. Αν είμαστε
          αρκετά κοντά, περνάμε στη State::DONE. */
    }
}
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Brownlee, Jason. Finite State Machines (FSM).
http://ai-depot.com/FiniteStateMachines/

[2] Champandard, Alex J. The Mechanics of Influence Mapping:
Representation, Algorithm & Parameters.
http://aigamedev.com/open/tutorial/influence-map-mechanics/

[3] Hansson, Niklas. Influence Maps I.
http://gameschoolgems.blogspot.com/2009/12/influence-maps-i.html

[4] Hansson, Niklas. Influence Maps II - Practical Applications.
http://gameschoolgems.blogspot.com/2010/03/influence-maps-ii-practical.html

[5] Kutsenok, Alex. Swarm AI: A General -
- Purpose Swarm Intelligence Technique.
http://mysite.verizon.net/resglap4/sitebuildercontent/sitebuilderfiles/SwarmAI04.pdf

[6] Lester, Patrick. A* Pathfinding for Beginners.
http://policyalmanac.org/games/aStarTutorial.htm

[7] Sakharov, Alexander. Finite State Machines.
http://sakharov.net/fsmtutorial.html

[8] Vassar College. CS331 lecture on Lexical Analysis.
http://cs.vassar.edu/~cs331/lectures/lexical-analysis.pdf

[9] Wikipedia. Genetic Algorithm.
http://en.wikipedia.org/wiki/Genetic_algorithm

[10] Wikipedia. StarCraft.
http://en.wikipedia.org/wiki/Starcraft