ΟΡΑΣΗ ΥΠΟΛΟΓΙΣΤΩΝ - ΕΡΓΑΣΙΑ 1η

Report//ΚΟΚΚΟΡΟΣ ΙΩΑΝΝΗΣ 57090

ΘΕΜΑ

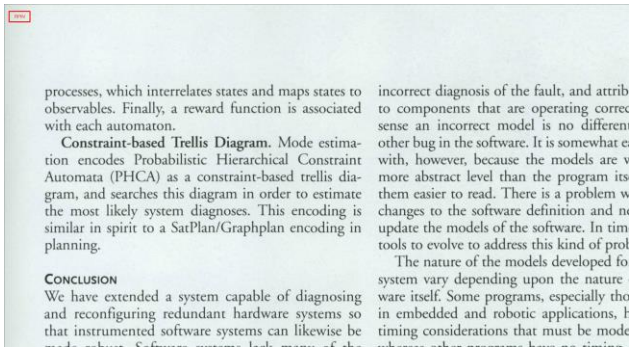
Να αναπτυχθεί μια ολοκληρωμένη μεθοδολογία και να υλοποιηθεί σε κώδικα, ώστε στις οκτώ ικόνες να επιτύχει τα εξής:

1. Ανίχνευση όλων των υπο-περιοχών κειμένου του εγγράφου. Ως υπο-περιοχή ορίζεται ένα τμήμα του εγγράφου με κείμενο, το οποίο διακρίνεται ως προς τη θέση του από τα υπόλοιπα μέρη του εγγράφου (π.χ. αρίθμηση σελίδας, υποσέλιδο, παράγραφος, τίτλος κ.α.). Το πρόγραμμα θα πρέπει να εμφανίζει και να αποθηκεύει την εικόνα εγγράφου στην οποία να είναι εμφανείς οι διαφορετικές υπο-περιοχές της. Συγκεκριμένα, για κάθε υπο-περιοχή να σχεδιαστεί ένα περιβάλλον κουτί (boundingbox) καθώς και ένας μοναδικός αύξων αριθμός.

2. Για κάθε μια υπο-περιοχή να μετρηθούν και να εμφανίζονται ως έξοδος του προγράμματος τα ακόλουθα μεγέθη:

a. Η επιφάνεια της υπο-περιοχής που καταλαμβάνεται από κείμενο, που ορίζεται ως ο αριθμός των εικονοστοιχείων που ανήκουν σε γράμμα (σκουρόχρωμα) και όχι στη σελίδα (ανοιχτόχρωμα)
b. Η επιφάνεια του περιβάλλοντος κουτιού(boundingbox) τηςυπο-περιοχής.
c. Ο αριθμός των λέξεωνπου εμπεριέχονται στην υπο-περιοχή
d. Η μέση τιμή διαβάθμισης του γκρι των εικονοστοιχείων που περιέχονται στα περιβάλλοντα κουτιά (boundingboxes) των αντικειμένων, με τέτοιο τρόπο ώστε η ταχύτητα εκτέλεσης υπολογισμού να είναι ανεξάρτητη του μεγέθους της υπο-περιοχής.

Καθώς οι αρχικές εικόνες ηταν πολύ μεγάλων διαστάσεων χρειάστηκε να ξεκινήσω με την δημιουργία της συνάρτησης ResizeWithAspectRatio() όπου δέχεται ως είσοδο μια εικόνα , ένα μήκος και ένα πλάτος . Αν οι τιμές των μήκος και πλάτος αντίστοιχα είναι μη μηδενικές τότε το πλάτος και το μήκος της εικόνας προσαρμόζεται στις νέες τιμές διατηρώντας την ανάλυση της εικόνας. Ακολουθούν οι έξοδοι της εικόνας πριν και μετά την χρήση της



processes, which interrelates states and maps states to observables. Finally, a reward function is associated with each automaton.

Constraint-based Trellis Diagram. Mode estimation encodes Probabilistic Hierarchical Constraint Automata (PHCA) as a constraint-based trellis diagram, and searches this diagram in order to estimate the most likely system diagnoses. This encoding is similar in spirit to a SatPlan/Graphplan encoding in planning.

Conclusion

We have extended a system capable of diagnosing and reconfiguring redundant hardware systems so that instrumented software systems can likewise be

incorrect diagnosis of the fault, and attrib
to components that are operating correc
sense an incorrect model is no differen
other bug in the software. It is somewhat e
with, however, because the models are v
more abstract level than the program its
them easier to read. There is a problem w
changes to the software definition and n
update the models of the software. In tim
tools to evolve to address this kind of prob

The nature of the models developed fo
system vary depending upon the nature
ware itself. Some programs, especially tho
in embedded and robotic applications, h
timing considerations that must be mode

ΕΙΚΟΝΑ 1

processes, which interrelates states and maps states to observables. Finally, a reward function is associated with each automaton.

Constraint-based Trellis Diagram. Mode estimation encodes Probabilistic Hierarchical Constraint Automata (PHCA) as a constraint-based trellis diagram, and searches this diagram in order to estimate the most likely system diagnoses. This encoding is similar in spirit to a SatPlan/Graphplan encoding in planning.

Conclusion

We have extended a system capable of diagnosing and reconfiguring redundant hardware systems so that instrumented software systems can likewise be made robust. Software systems lack many of the attributes of hardware systems to which the described methods have traditionally been applied; they tend to be more hierarchical and have more complex and numerous component interactions. Software components and their interconnections represent a significantly higher modeling burden.

Our approach differs from other similar techniques in the following ways:

- Models specify program behavior in terms of abstract states, which simultaneously makes the models easier to read and think about and somewhat robust to changes in low-level software implementation decisions.
- Modeling covers a wide spectrum of software considerations from a high-level storyboarding of the software to temporal considerations, if any, to the causal relationships between components.
- Robustness and recovery derives from a collection of complex and highly tuned reasoning algorithms that estimate state, choose contingencies, and plan state trajectories. The programmer is largely shielded from this complexity because the mechanism is hidden behind the intuitive unified modeling language.

An interesting feature of our approach is the ability to add robustness incrementally. More modeling leads to greater runtime robustness because it allows the system to detect, diagnose, and repair more fault situations. This means the effort devoted to modeling can be managed in much the same way that is done for test suite development in conventional software development projects.

Modeling errors can result in a number of undesirable outcomes such as failure to detect fault conditions and subsequent failure to recover from the fault, incorrect diagnosis of the fault, and attributing faults to components that are operating correctly. In this sense an incorrect model is no different from any other bug in the software. It is somewhat easier to deal with, however, because the models are written at a more abstract level than the program itself, making them easier to read. There is a problem with making changes to the software definition and neglecting to update the models of the software. In time we expect tools to evolve to address this kind of problem.

The nature of the models developed for a software system vary depending upon the nature of the software itself. Some programs, especially those involved in embedded and robotic applications, have critical timing considerations that must be modeled as such whereas other programs have no timing of synchronization considerations.

Developing model-based reconfigurable software systems is a relatively new endeavor, but results of our early experiments are encouraging. Much work remains to extend the current experimental system to cover the full range of software practice.

References

1. Bernard, D., Dorais, G., Gamble, E., et al. Spacecraft autonomy flight experience. *The DS1 remote agent experiment. In Proceedings of the AIAA Space Technology Conference and Exposition* (Albuquerque, NM, Sept. 1999).
2. Ben *RAP Language Manual*. Working Note AAP-6, University of Chicago, 1995.
3. Gat, E. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the AAAI Fall Symposium on Plan Execution* (Cambridge, MA, Nov. 1996).
4. Laddaga, R., Robertson, P., and Shrobe, H.E. Introduction to self-adaptive software: Applications. In *Proceedings of the 2nd International Workshop on Self-Adaptive Software (IWSAS 2001)* (Balatonfüred, Hungary, May 2001). LNCS 2614. Springer.
5. Mikaelian, T., Williams, B.C., and Sachenbacher, M. Diagnosing complex systems with software-extended behavior using constraint optimization. In *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX-05)* (Monterey, CA, June 2005).
6. Simmons, R. Structured control for autonomous robots. *IEEE Transaction on Robotics and Automation 10*, 1 (1994), 34–43.
7. Williams, B. and Nayak, P. A reactive planner for a model-based executive. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)* (Nagoya, Japan, August 1997).

Paul Robertson (paulr@mad.scientist) is a research scientist at the Massachusetts Institute of Technology's Computer Science and Artificial Intelligence Laboratory, Cambridge, MA.

Brian Williams (williams@mit.edu) is Boeing Associate Professor of Aeronautics and Astronautics at the Massachusetts Institute of Technology, Cambridge, MA.

ΕΙΚΟΝΑ 2

Στην συνέχεια για την φόρτωση της εικόνας στο πρόγραμμα χρησιμοποίησα cv2.imread() . Εφαρμόζω την συνάρτηση cv2.medianBlur για καθαρισμό του salt and pepper noise (δεν κατάφερα να φτιάξω δικό μου φίλτρο που να ικανοποιεί τις ανάγκες της εργασίας με αποτέλεσμα να μην επηρεάζεται και η επίδοση του προγράμματος ανάμεσα σε original και noised εικόνες).

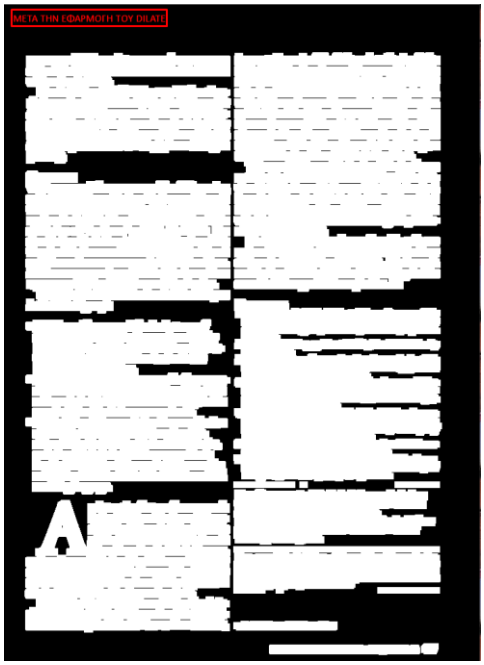
ΕΙΚΟΝΑ 3

ΕΙΚΟΝΑ 4

Έπειτα πρέπει η εικόνα να μετατραπεί σε Grayscale για αυτό τον λόγο χρησημοποιώ την έτοιμη συνάρτηση της OpenCV cv2.cvtColor(resize, cv2.COLOR_BGR2GRAY) . Για την επεξεργασία της εικόνας πρέπρει πρωτα να την μετατρέψω σε binary μορφή μέσω της συνάρτησης cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1] όπου κάθε pixel μετατρέπεται σε 0 αν έχει τιμή μεγαλύτερη του thresh αλλιώς παίρνει την τιμή 1. Χρησιμοποίησα την cv2.THRESH_OTSU για την εύρεση της κατάλληλης τιμής του thresh

ΕΙΚΟΝΑ 5 **ΕΙΚΟΝΑ 6**

Για την δημιουργία ενός ορθογώνιου kernel διαστάσεων 5χ4 καλώ την cv2.getStructuringElement(cv2.MORPH_RECT, (55, 4)) . Έπειτα πρέπει οι περιοχές που υπάρχει κείμενο να γίνουν πιο έντονες δηλαδή να αντικατασταθεί το μαύρο φόντο μεταξύ των σειρών και των λέξεων με λευκό για τον σχηματισμό countors. Αυτό γίνεται με την χρήση της cv2.dilate(thresh, kernel, iterations =3) όπου δέχεται ως εισόδους το thresh και το kernrel το οποίο το εφαρμόζει πάνω στην threshed εικόνα. Αν έστω και ένα pixel της περιοχής του kernel έχει τιμή ίση με 1 τότε όλο το kernel παίρνει την τιμή 1. Αυτό έχει ως αποτέλεσμα να αυξηθεί η περιοχή του άσπρου στην εικόνα. Η διαδικασία επαναλαμβάνεται 3 φορές.



ΕΙΚΟΝΑ 7

Για την εύρεση της κάθε παραγράφου χρησιμοποιήθηκε η cv2.findContours(dilate, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE) όπου δέχεται ως είσοδο την dilate εικόνα και

στην συνέχεια εντοπίζει όλα τα countors δηλαδή τις λευκές περιοχές. Κάθε countor αποτελεί υποθετικά και μια παράγραφο. Χρησιμοποίησα την cv2.RETR_EXTERNAL καθώς θέλω μόνο τις υψηλά ιεραρχικά περιοχές δηλαδή το περίγραμμα κάθε μιας. Η cv2.CHAIN_APPROX_SIMPLE αποθηκεύει μόνο τις γωνίες κάθε περιοχής γλιτώνοντας χρόνο και μνήμη.

Στην συνέχεια εκτελείται μια επανάληψη οπού μέσω της cv2.boundingRect() βρίσκω τις θέσεις τον γωνιών κάθε countor και με την cv2.rectangle(resize , (x, y), (x + w, y + h), (0, 0, 255), 1) ζωγραφίζονται τα περιγράμματα κάθε ενός με κόκκινο χρώμα και intesity ίσο με ένα. Για την αρίθμηση της κάθε υπο-περιοχής χρησιμοποιείται η cv2.putText(resize, str(i), (x+1, y+15), cv2.FONT_HERSHEY_COMPLEX, 0.5, (0, 0, 255), 1) όπου δέχεται ως είσοδο την εικόνα και τυπώνει τον αριθμός κάθε countor στην πάνω αριστερή γωνιά του με κόκκινο χρώμα και intesity ίσο με ένα.

Παρατηρείται ότι εμφανίζει σαν countors τις περιοχές 4, 8 , 9 ενώ δεν θα έπρεπε και αυτό γιατί στο dilate υπάρχει μια μικρή απόσταση μεταξύ των άσπρων περιοχών (βλέπε εικόνα 7)

ΕΙΚΟΝΑ 8

Η μεση τιμη του γκρι κάθε countor βρέθηκε με την συνάρτηση resize.mean(axis=0).mean() όπου βρίσκει την μέση τιμή του γκρι κάθε γραμμής και στην συνέχεια την νέα μέση τιμή. Το εμβαδόν κάθε περιοχής το βρήκα με τον πολλαπλασιασμό των μήκος και πλάτος κάθε περιοχής αντίστοιχα. Για την εύρεση κάθε λέξης ξανά εκτέλεσα την όλη διαδικασία με νέα μάσκα με διαστάσεις 1χ1. Στην συνέχεια σε κάθε επανάληψη εκτέλεσα μια νέα επανάληψη οπού μετρούσα κάθε στοιχείο του cnt2 . Αυτό είχε ως αποτέλεσμα της εύρεσης όλων των λέξεων του κειμένου και όχι της κάθε παραγράφου.

Σημείωση : Για τα παραδείγματα χρησιμοποιήθηκε η ‘2_noise.png’ εικόνα.