Testwell CTC++

Theo Lauverjat, Adrien Soler, Lucas Lemaire

Juin 2020

Table des matières

1 Etat de l'art sur les logiciels de tests

Parlons dans un premier des logiciels de tests dans leur ensemble. Quand on parle de test on parle en réalité d'un ensemble très large qui comporte de nombreuses ramifications. L'objectif d'un test est d'abord de valider la conformité d'un programme par rapport à des spécifications données.
On peut séparer les tests en deux catégories : statiques et dynamiques.

Dans une analyse statique, le code est examiné sans être exécuté alors que dans une analyse dynamique le code est exécuté et c'est cette exécution qui va être examiné.

Dans la catégorie des analyses statiques on peut distinguer les analyses manuelle et automatiques.
En l'occurrence les analyses manuelles sortent du cas de notre étude puisqu'on ne parle pas ici de logiciel de test mais de technique manuelle comme la revue ou l'inspection du code.

Parlons donc uniquement, pour le moment, des différences dans l'analyse statique automatique.
On peut distinguer plusieurs outils :
Premièrement les comparateurs, leur but est simple, ils permettent d'indiquer les différences entre deux fichiers. Cela peut se révéler particulièrement utiles lorsqu'il s'agit de contrôler les modifications d'une version à une autre (c'est un outil particulièrement utilisé par git)

On retrouve également les outils d'analyse statiques du code comme Lint qui permettent une détection des erreurs latentes liées aux typages, aux appels de fonctions, etc. Avec le temps ces types d'analyses sont de plus en plus intégrés directement dans le compilateur qui informe de ce type d'erreurs via des avertissements lors de la compilation.

On peut également retrouver l'élaboration de graphes de contrôles. Ceux-ci sont construits après examination du code et permettent une représentation algorithmique séquentielle de l'exécution du code (sans son exécution). Cela permet notamment des analyses du flot de données et ainsi de détecter les problèmes liés à l'ordre des manipulations de variables (utilisation d'une variable sans assignation préalable, assignation d'une variable sans utilisation ultérieure, etc.).

Enfin on peut également contrôler la qualité du code à travers un certains nombres de métriques. On sépare ces métriques en deux catégories. Tout d'abord il y a les métriques directs (les résultats sont directement observés à partir des chiffres obtenus) : quelques exemples sont le nombre de lignes du code, le nombre de commentaires, le nombre d'erreurs sur une période donnée, etc. Il y a aussi les métriques indirect (les résultats sont déduits d'un traitement des chiffres obtenus) comme la complexité cyclomatique qui mesure la complexité des décisions dans la structure d'un module (celle ci est déterminée à partir du graphe de contrôle)

L'analyse statique se reposait sur une analyse du code sans exécution, voyons maintenant la catégorie des analyses dynamiques qui elle se repose sur une exécution du code.
D'entrée de jeu on peut directement les séparer en deux catégories : les tests en boîte blanches et ceux en boîte noire.

Comme leur nom l'indique la différence se trouve dans la connaissance ou non de la structure interne du code.

Dans les tests en boîte blanche on sélectionne les données fournies en entrée par rapport à la structure interne du code. On cherche ici à valider le comportement d'une certaine partie du code. C'est dans cette catégories que l'on retrouve la plupart du temps les tests unitaires ainsi que les tests de couvertures.
Le test unitaire consiste en la sélection d'un jeu de données d'entrée et la vérification de la sortie. Si la sortie est celle attendu, le test est validée ; dans le cas contraire le test est non valide.
Le test de couverture vise à donner une information sur la proportion du code source utilisé pour un jeu de données. La couverture peut être une métrique très importante puisqu'elle informe l'éventuelle client sur la proportion du code testé et validé (une suite de tests ayant un faible taux de couverture aura beaucoup plus de chance de ne pas détecter un bug logiciel). La couverture peut être mesuré de plusieurs façons différentes : on peut ainsi mesurer la couverture des instructions, mais aussi celle des branchements ou encore celle des chemins (en pratique infaisable). En général une bonne couverture prend en compte un regroupement de ces différents critères.
Les tests en boîte blanche permettent de contrôler la robustesse du code source en sélectionnant des jeux de données inattendu. Il ne permettent cependant pas de détecter une éventuelle omission.

Pour le test en boîte noire, on sélectionne, cette fois, le jeu de données et les résultats attendus par rapport aux fonctionnalités attendus même du produit (et non par rapport à ce que le code est sensé faire).
On y retrouve par exemple les tests systèmes et les tests d'acceptation qu'on ne détaillera pas outre mesure.
Ces tests en boîte noire ont l'avantage de pouvoir détecter d'éventuelle omissions et d'être réutilisables entre les différentes version du code puisqu'ils ne se basent pas dessus. Ils ne donne cependant pas d'information sur la localisation d'une erreur si elle est detécté.

Ce tableau résume les différents type de tests:

Tests			
Statiques		Dynamiques	
Manuels	Automatiques	Boîte Blanche	Boîte Noire
Revue Inspection	Comparateur Graphes de contrôle Compilateur Métriques	Tests Unitaires Tests de couverture	Tests Système Test d'acceptation

2 Justification du choix du logiciel

Pour chercher quel logiciel de test utiliser nous avons tout d'abord pris sélectionné plusieurs logiciels dans une liste rencensant beaucoup de logiciels de test.

Ensuite, nous avons cherché concrètement comment se présentaient ces logiciels sélectionnés. Pour certains, aucune information n'était vraiment disponible et pour d'autres le logicielne semblait pas présenter de versions d'essai.

Quant à Testwell CTC++ : des vidéos youtube montraient son intégration dans Code::Blocks et Visual Studio, les deux plateformes de programmation dont nous étions familier.
La possibilité d'avoir un rapport html interactif nous a beaucoup plu. Sur ce html il est possible de naviguer à travers le code ou même d'aller directement aux fonctions.
De plus, Verifysoft a été très réactif lors de nos échanges par mail pour obtenir la version d'essai et quand nous avions des questions.
Nous les remercions encore pour leur version d'essai et leur temps accordé.

Pendant nos recherches nous avons bien sûr vu le logiciel de test de Google. Cependant nous avons vite mis de côté ce logiciel en accord avec le mouvement "Dégooglisons Internet" : En effet nous sommes déjà beaucoup trop dépendant de cette société, c'est pourquoi nous pensons que présenter des alternatives viables est une démarche importante, plutôt que de céder à la facilité.

3 Présentation du logiciel

Testwell CTC++ Test Coverage Analyzer est un outil de couverture de test(Boite blanche) pour les projets en C, C++, C et Java.
Testwell CTC++ gère tous les types de couverture:

- Couverture des Déclarations (Statement Coverage)
- Couverture de Fonction (Function Coverage)
- Couverture des conditions (Condition Coverage)
- Couverture de Décision / Couverture de branche (Decision Coverage / Branch Coverage)
- Couverture de Condition multiple (Multicondition Coverage).

Testwell CTC++ comprend aussi un analyseur de temps d'execution.

4 Installation et Utilisation

4.1 Téléchargement CTC++

Testwell CTC++ est un framework professionnel qui recquiert une licence payante. Ainsi il est nécessaire d'acheter le logiciel ou de demander une version d'essai pour pouvoir accéder à l'espace de téléchargement.

4.2 Installation CTC++

Executer l'installateur "setup.exe". Rajoutez le fichier testwell.lic fourni par l'entreprise VerifySoft dans le dossier CTC (il faut que la licence soit valide et non expiré).

4.3 Configuration de L'IDE

4.3.1 Code::Blocks

Dans Settings → Compiler → Toolchain executables,
Changez les champs suivants avec les valeurs ci dessous.

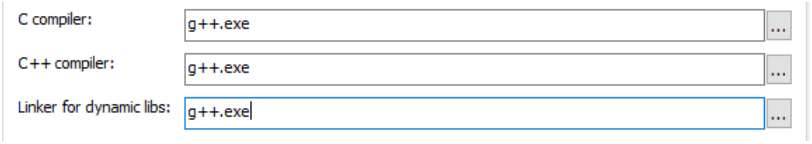

Figure 1: Choix Compilateur

Dans Settings → Compiler → Custom variables, Ajoutez une Variable en cliquant sur ADD.
Puis completez les champs suivants:

Cliquez sur OK.

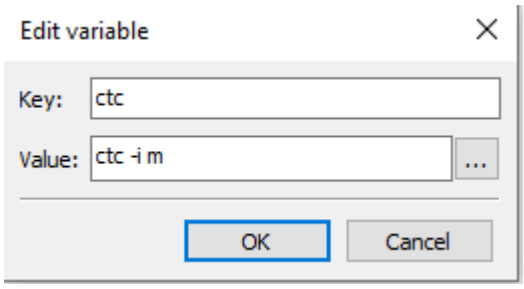

Figure 2: Custom variable

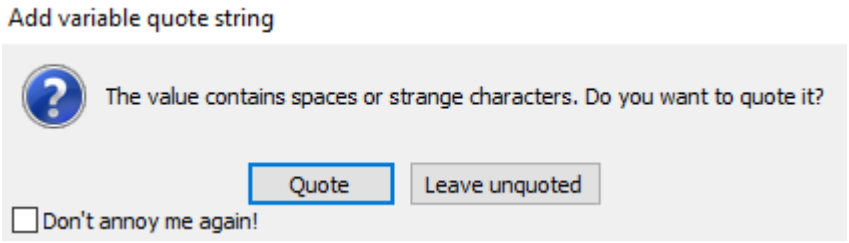

Figure 3: Unquote

Puis Appuyez sur "leave unquoted".

Dans Settings → Compiler → Other settings, Cliquez sur Advanced options, appuyez sur yes.
Rajoutez $(ctc) dans Command line macro pour chaque attribut dans la liste déroulante Command.

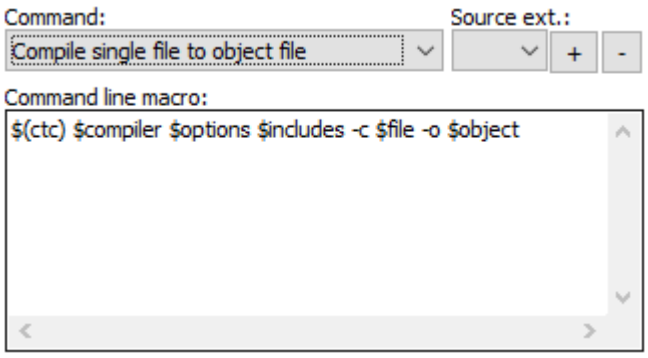

Figure 4: command line macro

Coverage Report Dans Tools → Configure tools,
Ajoutez les 2 outils avec le bouton ADD et en complétant les champs ci-dessous.
Puis, dans Tools → generate textual report.(Figure 5)
Puis, dans Tools → generate html report.(Figure 6)

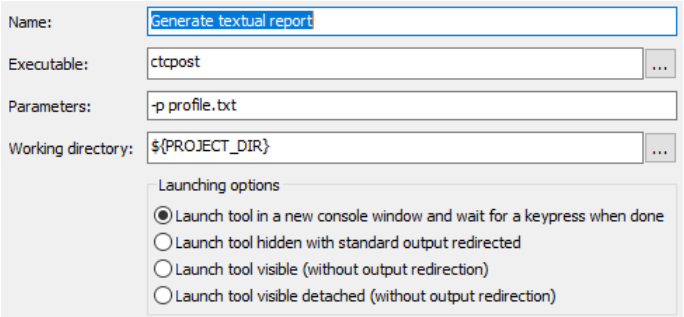

Figure 5: textualreport

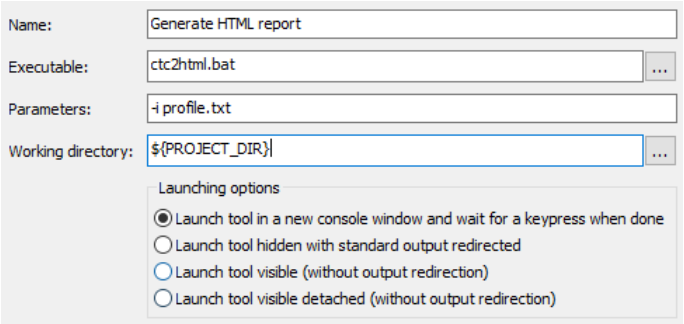

Figure 6: htmlreport

Time execution Report Dans Tools → Configure tools,
Ajoutez GenerateTextualTimeReport avec le bouton ADD et en complétant les champs ci-dessous.(Figure 7)

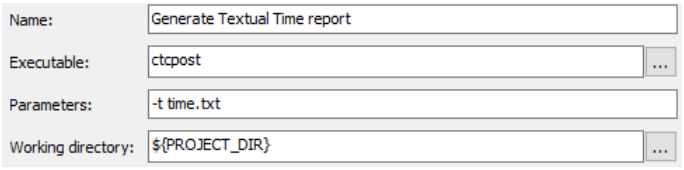

Figure 7: GenerateTextualTimeReport

Execution Pour générer le rapport html il faut "Build" le programme. Lancer le programme (une ou plusieurs fois) avec des jeux de test.
Dans Tools, utiliser "Generate textual report" puis "Generate HTML report". On peut aussi utiliser "Generate Textual Time Report" pour créer un fichier time.txt dans le dossier du projet qui permet d'analyser le temps passé dans chaque fonction.

4.3.2 Visual Studio

L'intégration sur Visual Studio est beaucoup plus simple. En effet, elle est prévue directement par CTC++.

Ouvrez un invité de commande et tapez :

" cd %CTCHOME%\Vs_integ "

Vous devez arriver dans le dossier du logiciel. Si ce n'est pas le cas, cherchez dans un explorateur de fichier le chemin d'accès du fichier Vs_integ. Si vous n'avez pas changé les paramètres par défaut, sur Windows il ressemble à :
C:\Testwell\CTC\Vs_integ tapez donc " cd <chemin>" sans les < >.

Ensuite, tapez :

vs_integ.bat -install

Un programme s'exécute et vous demande de confirmer l'installation : "OK to do the install? [Y/n]". Tapez " Y " et validez.

Une fois l'installation effectuée, fermez l'invité de commande et ouvrez Visual Studio.

Ouvrez un projet puis cliquez sur Outils :

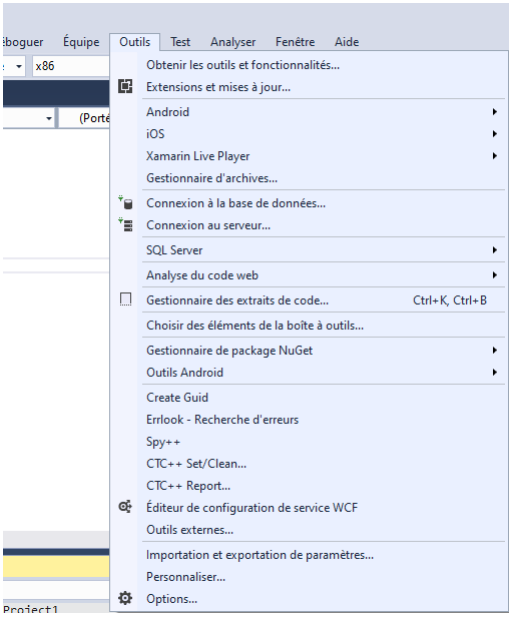

Figure 8: Outils Visual Studio

Vous devriez remarquer l'apparition de CTC++ Set/Clean... et de CTC++ Report...
Contrairement à Code::Blocks, les outils sont directement intégrés et bien plus complet.

Choisissez d'abord CTC++ Set/Clean...

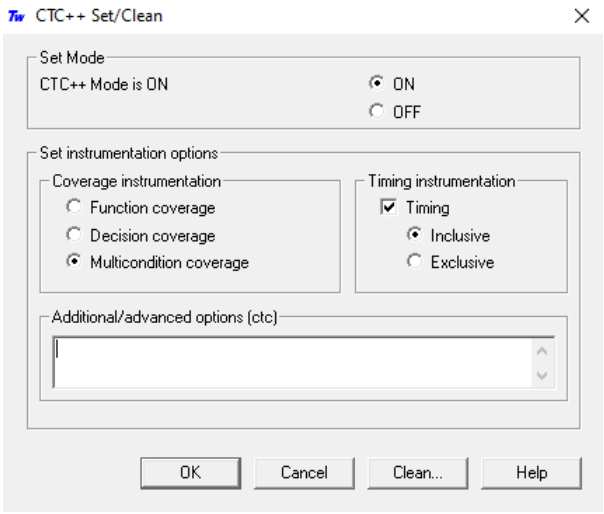

Figure 9: CTC++ Set/Clean

Une nouvelle fenêtre s'ouvre.
Tout d'abord dans Set Mode vous pouvez activer et désactiver le logiciel CTC++ à tout moment ici.

Dans Set Instrumentation options vous pouvez choisir les données à analyser.
Par défaut choisissez Multicondition qui regroupe les autres.
Dans Timing instrumentation vous pouvez choisir d'enregistrer les temps d'executions.
Inclusive et Exclusive sont expliqués dans la section Time Report.

Enfin le bouton Clean... permet de supprimer toutes les précédentes données qui ont pu être générées par CTC++ sur ce projet.
Cliquez sur OK.

Sur Visual Studio cliquez sur Générer → Regénérer nomProjet.

Ensuite Choisissez l'outil CTC++ Report...

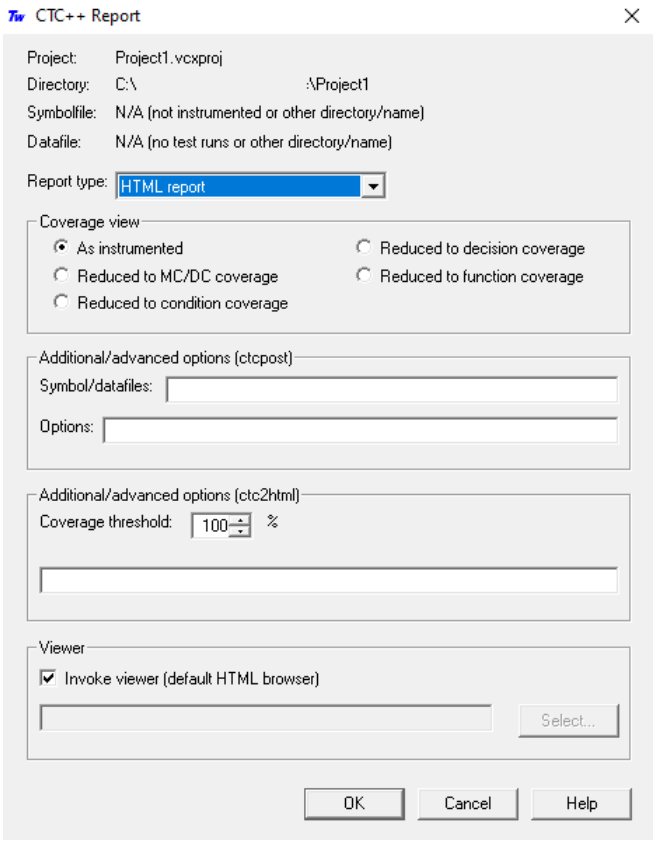

Figure 10: CTC++ Report

Sur cette nouvelle fenêtre vous pouvez tout d'abord choisir le type de rapport que vous voulez générer. "Consultez la section Mesures de tests fournis par le logiciel" pour connaître les spécificités de chacun.

Vous pouvez régler le pourcentage de Coverage threshold afin d'afficher en rouge toutes les fonctions dont moins de ce pourcentage a été exécuté.
Cliquez sur OK.
En fonction du type de rapport sélectionné, un rapport sera généré.

5 Mesures de tests fournis par le logiciel

Le rapport html contient:

5.1 Taux de Couverture du programme

Montre les différents taux de couverture en fonction des fichiers du projet.(Figure 11) A gauche, le taux de couverture des points de tests(Condition Coverage). A droite, le taux de couverture des instructions (Statement Coverage)

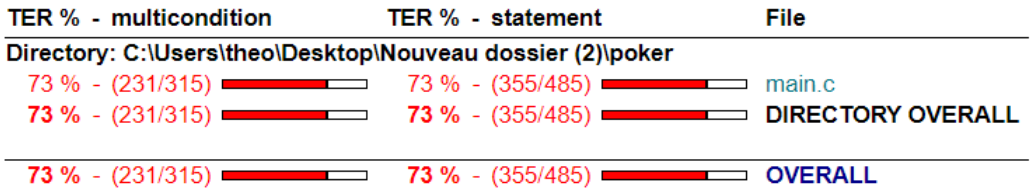

Figure 11: coverageSummary

5.2 Taux de Couverture des fonctions

Montre les différents taux de couverture en fonction des différentes fonction du projet.(Figure 12)

TER % - multicondition		TER % - statement		Calls	Line	Function
100 %	(4/4)	100 %	(4/4)	10	12	jeuNeuf()
87 % -	(7/8)	83 % -	(5/6)	10	19	affichegagnant()
100 %	(4/4)	100 %	(8/8)	50	37	melangerJeu()
100 %	(2/2)	100 %	(1/1)	90	47	carteSuivante()
100 %	(2/2)	100 %	(1/1)	90	52	affichercartes()
100 %	(8/8)	100 %	(16/16)	22	56	trihauteur()
100 %	(4/4)	100 %	(7/7)	116	83	Initzero()
77 % -	(21/27)	87 % -	(21/24)	14	94	IsPair()
84 % -	(11/13)	77 % -	(14/18)	18	136	IsBrelan()
66 % -	(8/12)	57 % -	(8/14)	4	166	IsCarre()
100 %	(17/17)	100 %	(17/17)	4	190	IsFull()
91 % -	(11/12)	95 % -	(19/20)	2	214	IsQuinteFlush()
100 %	(8/8)	100 %	(14/14)	20	240	retirerdoublon()
69 % -	(16/23)	76 % -	(13/17)	20	259	IsSuite()
100 %	(10/10)	100 %	(24/24)	20	290	IsCouleur()
100 %	(4/4)	100 %	(9/9)	2	325	Hauteur()
92 % -	(24/26)	100 %	(25/25)	20	337	Force()
90 % -	(10/11)	100 %	(9/9)	10	386	compare()
100 %	(10/10)	100 %	(28/28)	10	409	gagnant()
0 % -	(0/11)	0 % -	(0/9)	0	445	ScanMain()
0 % -	(0/8)	0 % -	(0/12)	0	461	Supprimercarte()
0 % -	(0/29)	0 % -	(0/32)	0	480	initCarteParHautCoul()
94 % -	(36/38)	100 %	(54/54)	520	517	initCarteParId()
100 %	(4/4)	100 %	(15/15)	10	578	DealBoard()
100 %	(4/4)	100 %	(6/6)	20	603	DealPocket()
37 % -	(6/16)	38 % -	(37/95)	1	616	main()
73 % - (231/315)		**73 %** - (355/485)				main.c
73 % - (231/315)		**73 %** - (355/485)				DIRECTORY OVERALL

Figure 12: fonctiontest

5.3 Code non testé

Ce rapport nous montre uniquement les parties de code non utilisées.
Les colonnes Hits/True et False nous informent du nombre de fois que l'instruction ou la condition a été atteinte et de la valeur booléenne de la condition.
Ce rapport nous montre aussi les occurences des valuations des conditions complexe.

TER: 73 % (231/315) structural, 73 % (355/485) statement

Hits/True	False	-	Line	Source
10			19	FUNCTION affichegagnant()
0	10	-	31	if (gagnanttamp == - 1)
14			94	FUNCTION IsPair()
	0	-	102	2: T && F
0	1	-	110	if (res . valeurs [1] == hero . carte [5] . mothauteur && res . valeurs [1] == hero . carte [6] . mothauteur)
0		-	110	1: T && T
	0	-	110	3: F && _
0	0	-	112	if (res . valeurs [2] == hero . carte [4] . mothauteur)
18			136	FUNCTION IsBrelan()
0	4	-	147	if (i == 4)
0	4	-	152	else if (i == 3)
4			166	FUNCTION IsCarre()
0	16	-	173	if (hero . carte [i] . mothauteur == hero . carte [i + 1] . mothauteur && hero . carte [i + 1] . mothauteur == hero . carte [i + 3] . mothauteur)
0		-	173	1: T && T && T
0	0	-	178	if (hero . carte [i] . mothauteur == hero . carte [6] . mothauteur)
2			214	FUNCTION IsQuinteFlush()
0	2	-	235	if (resnew . valeurs [0] != 0)

Figure 13: UntestedCode

5.4 Profil d'execution

En vert, les lignes qui ont été éxécutées au moins une fois.
En rouge, les lignes qui n'ont jamais été éxecutées.
En gris, les tests de conditions multiples.

Hits/True	False	Line	Source

		84	{
			valeur zera=*zero;
			int i;
			for(i=0; i<6; i++)
			{
			zera.valeurs[i]=0;
			}
			return zera;
			}

Top			

14		94	valeur IsPair(showdown hero) //100 %
			{
			valeur res;
			res=Initzero(&res);
			int i;
			int j=2;
			for(i=5; i>-1; i--)
			{
			if(hero.carte[i].hauteur==hero.carte[i+1].hauteur && res.valeurs[0]<2)
			1: T && T
			2: T && F
			3: F && _
			{
			res.valeurs[0]=res.valeurs[0]+1;
			res.valeurs[res.valeurs[0]]=hero.carte[i].mothauteur;
			}
			}
			if (res.valeurs[0]==2)
			{
			if(res.valeurs[1]==hero.carte[5].mothauteur && res.valeurs[1]==hero.carte[6].mothauteur)
			1: T && T
			2: T && F
			3: F && _
			{
			if(res.valeurs[2]==hero.carte[4].mothauteur)

Figure 14: execution profile

5.5 Time Report

Il existe deux types de calcul de temps:

- Inclusive = Le temps inclusif est calculé en comptant le temps passé dans la fonction ainsi que dans les fonctions appelé par cette fonction
- Exclusive = Le temps exclusif est calculé en comptant seulement le temps passé à l'intérieur de la fonction sans compter le temps des fonctions appellées.

Dans le rapport(Figure 15), on peut voir de gauche à droite le nombre d'exécution de la fonction, le total de temps passé dans la fonction(en ms), le temps moyen passé dans la fonction(en ms),le temps maximum passé dans la fonction (en ms), la ligne où se trouve le début de la fonction et son nom.

Si on veut plus de précision dans les timings, on peut faire passer en argument la fonction de calcul du temps que l'on souhaite utiliser.

```
*****************************************************************************
*         CTC++, Test Coverage Analyzer for C/C++, Version 9.1.0            *
*                                                                           *
*                          EXECUTION TIME LISTING                           *
*                                                                           *
*                  Copyright (c) 1993-2013 Testwell Oy                      *
*             Copyright (c) 2013-2020 Verifysoft Technology GmbH            *
*****************************************************************************


Symbol file(s) used   : MON.sym (Sun Jun 14 16:27:06 2020)
Data file(s) used     : MON.dat (Sun Jun 14 16:27:13 2020)
Listing produced at   : Sun Jun 14 16:27:22 2020
Execution cost type   : Clock ticks


MONITORED SOURCE FILE : C:\Users\theo\Desktop\Nouveau dossier (2)\poker\main.c
INSTRUMENTATION MODE  : multicondition+exclusive_timing

 EXECUTION      ========EXECUTION COST========
     COUNT      TOTAL      AVERAGE        MAX  LINE FUNCTION
=============================================================================
    100000        760          0.0          2    12 jeuNeuf()
    100000          2          0.0          1    19 affichegagnant()
    500000        621          0.0          2    37 melangerJeu()
    900000         15          0.0          1    47 carteSuivante()
    900000         26          0.0          1    52 affichercartes()
    206116         44          0.0          1    56 trihauteur()
   1194665         47          0.0          1    83 Initzero()
    169197         35          0.0          1    94 IsPair()
    184696         27          0.0          1   136 IsBrelan()
     15499          2          0.0          1   166 IsCarre()
     15152          4          0.0          1   190 IsFull()
      6116          5          0.0          1   214 IsQuinteFlush()
    200000         12          0.0          1   240 retirerdoublon()
    200000         82          0.0          2   259 IsSuite()
    200000         92          0.0          1   290 IsCouleur()
     34828          5          0.0          1   325 Hauteur()
    200000        204          0.0          2   337 Force()
    100000          2          0.0          1   386 compare()
    100000        101          0.0          1   409 gagnant()
         0          0          0.0          0   445 ScanMain()
         0          0          0.0          0   461 Supprimercarte()
         0          0          0.0          0   480 initCarteParHautCoul()
   5200000        214          0.0          2   517 initCarteParId()
    100000        127          0.0          2   578 DealBoard()
    200000        143          0.0          1   603 DealPocket()
         1       3112       3112.0       3112   616 main()
```

Figure 15: time report

6 Cas d'étude

6.1 Etude Préliminaire

On veut tester la couverture du programme et sa bonne exécution avec ce jeu de données :

- p 100000
- o 1000 2 0 3 1 4 0 5 1
- i 6 0 7 0 8 0 A 0 T 0 9 0 J 1 Q 1 K 1

On obtient une couverture de plus de 95 % (voir figure 16). De part la nature aléatoire du programme, si on réutilisait les mêmes jeux d'entrée pour déterminer la couverture on pourrait obtenir des résultats légérement différents. En regardant les fonctions n'atteignant pas 100% de couverture on remarque dans:

- Force(), que dans if (q res.valeurs[0]¡7) le deuxième argument est toujours vrai, en regardant le programme on se rend compte que cette vérification est bien inutile puisqu'à ce stade de la fonction une valeur >=7 est impossible.
- ScanMain(), que seul les cas valides ont été testés ainsi si il y a des erreurs lors de la saisie d'un jeu de données incohérent, les données de tests proposées ne peuvent pas garantir le bon fonctionnement du programme. Ainsi le logiciel nous amène a reconsidérer nos données de tests.
- initCarteParId() et initCarteParHautCoul(), que dans les cases le champ default n'a pas été spécifié ce qui peut poser des problèmes d'arrêt de programme. Cette erreur qui est normalement détectée lors des tests statiques est probablement détectée par l'analyseur multiconditionnel compris dans CTC++.
- Main(), que les jeux de tests ne testent pas l'appel du programme en lui transmettant les arguments en paramètre. De la ligne 711 à 730 on remarque une partie du code qui n'a jamais été éxécutée. En regardant de plus près on s'aperçoit que cette fonction n'est pas fonctionnelle. Ainsi il faudrait supprimer cette partie du code.

Malgrès 100000 itération du sous programme(p 100000), on remarque que la majorité du temps d'exécution (3112 ms) provient du main() ce qui correspond à l'entrée manuelle du jeu de test. Un résultat bien plus surprenant est que les fonctions d'initialisation du deck c'est à dire jeuNeuf() et MelangerJeu() prennent 2 tiers du temps d'execution(1381 ms). On peut en déduire que pour accélérer l'exécution du programme il suffit de chercher une meilleure initialisation des decks.

6.2 Test de vérification

6.2.1 Détermination du vainqueur

Cas	Input	Valeur attendu	Valeur trouvé	Valeur cohérente
Hauteur	i A 1 5 3 2 0 K 0 3 0 Q 1 J 3 8 0 7 3	Win :1	Win :1	oui
Paire	i 5 1 A 0 5 2 6 0 A 1 J 0 9 0 8 1 2 3	Loose :1	Loose :1	oui
Double Paire	i 5 1 A 0 6 2 A 2 5 2 J 0 9 0 8 1 6 3	Win :1	Win :1	oui
Brelan	i 9 1 A 0 9 2 A 2 5 2 J 0 9 0 8 1 6 3	Win :1	Win :1	oui
Quinte	i A 1 8 0 7 0 9 1 7 2 A 3 T 2 6 0 2 2	Loose :1	Loose :1	oui
Couleur	i 6 1 A 0 7 1 6 0 A 1 J 0 9 1 8 1 2 3	Win :1	Win :1	oui
Full	i 6 1 A 0 6 2 8 0 9 1 J 0 9 2 8 1 6 3	Win :1	Win :1	oui
Carre	i Q 0 J 0 5 2 J 1 J 2 J 3 8 0 T 2 5 0	Loose :1	Loose :1	oui
Quinte Flush	i 6 2 J 0 5 2 J 1 J 2 J 3 7 2 4 2 3 2	Win : 1	Win : 1	oui
Egalite Hauteur	i A 1 A 0 5 3 2 0 K 0 Q 1 J 3 8 0 7 3	Draw : 1	Draw :1	oui
Quinte à l'as	i A 1 K 0 T 0 8 1 2 0 3 3 4 2 5 1 8 0	Win : 1	Win : 1	oui
Quinte à l'as contre quinte supérieur	i A 1 6 0 K 0 8 1 2 0 3 3 4 2 5 1 8 0	Loose :1	Loose :1	oui

Figure 16: Jeu de test pour la fonction "détermination du vainqueur"

On obtient une couverture d'environ 95% pour les fonctions concernées par le test : scanMain(), initCarteParHautCoul(), gagnant() et les sous fonctions de gagnant(). Ainsi le jeu de test est un échantillon représentatif.
Tous les tests ont donné les résultats attendus.

6.2.2 Détermination de l'équité

Cas	Input	Valeur attendu[1]	Valeur trouvé	Valeur cohérente[2]
2 cartes supérieurs à une paire	o 100000 A 0 K 1 T 0 T 1	43290 480 56240	43471 441 56088	oui
2 cartes inférieur	o 100000 8 0 9 0 T 2 J 3	37740 1100 61150	37965 1028 61007	oui
Une carte supérieur, une carte inférieur	o 100000 K 0 2 0 J 2 Q 3	57990 700 41320	58363 685 40952	oui
Une paire inférieur à une autre	o 100000 2 0 2 1 T 0 T 1	17150 640 82210	17092 378 82530	non
1 cartes supérieurs à une paire	o 100000 Q 0 Q 1 A 2 5 2	66240 380 33380	66380 195 33425	non
2 cartes inférieur à une paire	o 100000 8 0 6 1 T 0 T 1	14600 500 84890	14664 424 84912	non

Figure 17: Jeu de test pour la fonction "détermination de l'équité"

1. Les valeurs attendus sont générées grâce au logiciel equilab, le chiffre des unités n'est pas significatif.

2. Test de bernoulli bilatéral avec un intervalle de confiance de 95%

La couverture est supérieur à 95% dans les fonctions autres que la fonction main() et de 100% dans la partie "détermination de l'équité".
Ainsi le jeu de test est un échantillon représentatif.

On remarque qu'il y a une erreur dans la détermination des égalités.

6.3 Avis sur l'utilité de CTC++ pour ce programme

Cet utilitaire de test se prête particulièrement bien au programme analysé en tant que test "boite blanche".
En effet, un test "boîte noire" nous aurait que difficilement permis l'analyse des problèmes du programme de part la nature aléatoire de ce dernier. Néanmoins il aurait été nécessaire d'utiliser préalablement un analyseur statique pour permettre de vérifier les bonnes pratiques du code en C (ce qui n'est pas le cas dans ce programme).

7 Avantages et faiblesses du logiciel

Avantages:

- Analyse de tous les niveaux de couvertures
- Clarté des rapports d'exécution
- Multi-Language (C,C++,C#,Java)
- Multi-OS (Windows, Linux, Solaris, HP-UX, OS X, AIX)
- Intégration rapide dans beaucoup d'environnements de développement
- Prise en main relativement simple
- Documentation d'utilisation complète et explicite
- Suit les normes de sécurité et kit de qualification(Annexe 1)
- Programme de test indépendant du programme testé
- Possibilité de tester le logiciel
- Très bonne disponibilité de l'équipe de communication : réponse en moins de 24 heures

Faiblesses:

- Licence payante
- Peu de contenu multimedia d'aide, pas de forum d'aide
- Impossible d'isoler la couverture d'une itération du programme lors d'un test sur plusieurs entrées

Notes :

- (*) : VerifySoft possède un outil de test unitaire mais celui-ci est limité. D'après une enquête auprès de ses clients et utilisateurs de Testwell CTC++, ces derniers utilisent plutôt des outils développés eux-mêmes ou bien des logiciels gratuits.
  Un point positif que l'on peut attribuer à l'entreprise est de ne pas chercher à tout prix à vendre des solutions qui ne seraient pas totalement adaptées. Ainsi, sur leur propre site, ils donnent notamment une liste d'outils Open Source de tests unitaires compatibles avec TestWell CTC++ : https://www.verifysoft.com/fr_ctcpp_unit_testing.html

8 Conclusion

Testwell CTC++ est un logiciel très facile et pratique d'utilisation :

Une fois installé et après une rapide intégration dans notre outil de programmation habituel, il permet de tester son code et d'obtenir un rapport détaillé et interactif.
Le rapport temporel est aussi un plus non négligeable quand on cherche à étudier rapidement la complexité de son programme.

Pourquoi utiliser un logiciel de couverture de code quand on peut directement utiliser un logiciel de test unitaire ? Car tester le jeu de valeur complet peut prendre beaucoup trop de temps. Grâce à une couverture de test on peut réduire les tests à quelques parties de code.
De plus seule une couverture de test permet de détecter du code mort.

Testwell CTC++ est un logiciel payant. Pour une utilisation étudiante il existe des solutions gratuites.
Cependant pour une utilisation professionnelle il est très intéressant, cela assure notamment la fiabilité du logiciel et du support derrière.
Le fait qu'il soit utilisé par des compagnies travaillant dans l'aviation permet d'être sûr que les tests sont fiables.

9 Annexe

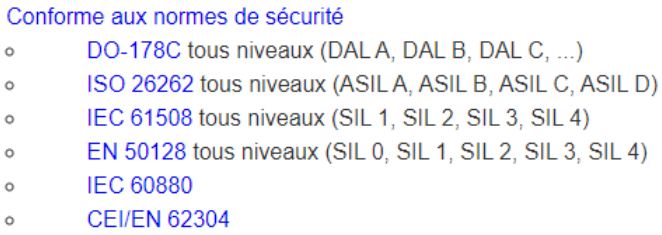

Figure 18: Conforme aux normes de sécurité