دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

گزارش فاز ۲ پروژه درس آزمون پذیری

(طراحی و پیاده‌سازی الگوریتم تولید بردار تست D، فشرده سازی بردارهای تست و مقایسه نتایج)

نام اعضای گروه :

علی فراهانی - ۴۰۰۲۱۱۰۸۷

آوا دژبان - ۴۰۰۲۱۱۱۶۲

نام استاد: دکتر شاهین حسابی

تابستان ۱۴۰۱

فهرست مطالب

۱. تعریف پروژه

در این فاز قصد داریم تا پیاده سازی الگوریتم تولید بردارهای تست (الگوریتم D) را شرح دهیم. جهت پیاده سازی این بخش، ابتدا طبق روش هم ارزی اشکال ها و غلبه اشکال، تعدادی از اشکال های ذکر شده در فایل isc. مدارهای ترکیبی را کاهش داده و سپس وارد فاز عملیاتی الگوریتم D میشویم.

پس از بدست آوردن بردارهای تست و اشکالاتی که توسط این بردارها کشف میشوند، وارد مرحله فشرده سازی بردارها شده و نشان میدهیم چگونه میتوان بوسیله این کار، تعداد زیادی از بردارهای تولید شده در مرحله قبل را کاهش داد. این کاهش برداری موجب میشود تا در اجراهای بعدی زمان زیادی ذخیره گردد.

در نهایت، به مقایسه الگوریتم D و الگوریتم تست جامع میپردازیم و با اعداد و آماری که گزارش میکنیم، متوجه تفاوت این دو الگوریتم شده و همچنین پارامترهای دیگر را نیز تحلیل خواهیم نمود.

۱.۱. نیازمندی های پروژه :

- پیاده سازی بخش کاهش اشکالات موجود در مدار بوسیله الگوریتم هم ارزی اشکال ها و غلبه اشکال.
- پیاده سازی الگوریتم D و بدست آوردن تمامی بردارهای تست مربوط به اشکالات باقی مانده.
- پیاده سازی فشرده ساز بردارهای تست.
- مقایسه الگوریتم D و روش تست جامع از نظر زمان و سایر پارامترها.

*** توجه مهم: ساختار فایل ها و پوشه های پروژه در آخرین صفحه آمده است.

۲. توضیحات کد

۲.۱. پیاده سازی روش هم ارزی و غلبه اشکال

جهت یافتن اشکالات هم ارز و قابل کاهش، ابتدا توسط تابع read_circuit_file (که در فاز اول نیز استفاده شد)، مشخصات و مقادیر مدار را از فایل isc. مدنظر میخوانیم. سپس از این مشخصات جهت بدست آوردن اشکالات هم ارز استفاده نموده و آنرا به عنوان پارامتر ورودی برای تابع equivalence_faults_check در نظر میگیریم.

در این تابع، به دنبال نود هایی هستیم که در فایل مدار به عنوان گیت منطقی آورده شده اند (در اینجا شاخه fanout یا ورودی بودن مهم نیست). سپس با یک حلقه، به ازای هر گیت، ورودی های آن را بدست آورده و با توجه به نوع گیت، هم ارزی و غلبه اشکال را اعمال میکنیم. برای مثال، برای یک گیت AND دو ورودی، مقادیر Sa0 ورودی ها را حذف کرده و تنها Sa0 در خروجی را نگه میداریم.

در اینجا از روش forwarding استفاده کرده ایم. بدین معنی که اشکالات حذف شده در ورودی ها اتفاق افتاده و اشکال هم ارز باقیمانده در خروجی میباشد. به همین دلیل، ورودی هایی که Sa0 یا Sa1 آنها طبق قانون هم ارزی حذف میشود، بجای پارامتر اشکال آنها در فایل isc. ، مقدار *** قرار میگیرد که نشان دهنده اشکالات حذف شده است.

۲.۲. پیاده سازی الگوریتم D

پس از یافتن اشکالات هم ارز و حذف آنها، وارد فاز اصلی الگوریتم D میشویم. پیاده سازی این الگوریتم (همانند روش اصلی مطرح شده در فصل ۱۱) دارای چهار مرحله اصلی است:

۱) ایجاد مقدار مناسب در محل اشکال در نود مد نظر: Primitive D-Cube
۲) انتشار اشکال در طول مدار و رساندن آن به یکی از خروجی ها: D-Drive/Propagation D-Cube
۳) ارزیابی/سنجش درستی مقادیر داده شده به نودها در مرحله ۲: Consistency/Justification
۴) عقب گرد و اعمال مقادیر متفاوت به ورودی ها در صورت نیاز: Backtrack

پایان الگوریتم D، زمانی است که اشکال مدنظر در صورت امکان در **تمام خروجی های مدار** دیده شده و مقادیر اعمال شده به نودها دارای consistency بوده و با یکدیگر تداخلی نداشته باشند. در نهایت، مقادیر

بدست آمده برای ورودی ها (بردارهای تست) و اشکالاتی که هر بردار کشف میکند، در پوشه fault در فایلی با فرمت fault_circuit_name.txt ذخیره میشود.

در کد پیاده سازی شده، به ازای هر کدام از این ۴ مرحله، یک تابع و چندین متغیر در نظر گرفته شده است که در ادامه به آنها میپردازیم:

۲.۳. ایجاد مقدار در محل اشکال (Primitive D-Cube)

در فایل اصلی (main.py)، پس از بدست آوردن اشکالات هم ارز و حذف آنها، تابع pdf فراخوانی میشود. این تابع سه پارامتر ورودی شامل مشخصات بدست آمده از فایل isc. مدار، ورودی های داده شده به مدار و نام مدار را گرفته و شروع به کار میکند. این تابع به ازای هر مدار تنها یکبار فراخوانی میشود.

درون این تابع، بوسیله یک حلقه، به ازای هر نود در مدار، بررسی میکنیم که این نود دارای کدام یک از اشکالات Sa0، Sa1 یا هر دو اشکال بطور همزمان است. این حلقه باعث میشود تا تمام نودها در مدار مدنظر بررسی شوند و بردار تست برای اشکالات تمام نودها بدست آید.

اگر بر روی یک نود، اشکال Sa0 قرار داشت، بجای اشکال آن مقدار d_f (به معنای ۱/۰) و اگر اشکال Sa1 قرار داشت، مقدار d_not_f (به معنای ۰/۱) میگذاریم. سپس این مقدار را به همراه مشخصات آن نود و مشخصات کل مدار به تابع fault_prop ارسال میکنیم تا مسیر انتشار خطا بدست آید.

۲.۴. انتشار اشکال در مدار (D-Drive)

هدف این بخش، یافتن مسیر انتشار خطا و رساندن آن به خروجی های مدار است. از آنجایی که انتشار خطا میبایست از مسیر D-frontier انجام گردد، این قسمت شامل تابع fault_prop و d_frontier میباشد. در تابع fault_prop، بوسیله حلقه while بررسی میشود که آیا همچنان یک گیت در مسیر D-frontier وجود دارد تا بتوان خطا را از آن گیت منتشر کرد یا خیر. تا هر زمان که این مسیر موجود باشد، وارد حلقه شده و بررسی میکنیم که گیت مد نظر از چه نوعی است. سپس، از روی ورودی کنونی (مقدار اشکالی که به آن رسیده)، ورودی دیگر را بدست می آوریم. برای مثال اگر اشکال به گیت AND رسیده باشد، ورودی یا ورودی های دیگر آن باید نقیض مقدار کنترلی یعنی ۱ باشند تا اشکال بتواند انتشار یابد. مقادیر اعمال شده به نودها در طول مرحله انتشار در دیکشنری prop_matrix ذخیره میگردند.

همچنین، اگر مقداری در محل یک شاخه fanout پدیدار شود، باید شاخه های دیگر و خطی که از آن مشتق شده نیز مقدار متناسب با آن بگیرند. مثلا اگر یک شاخه مقدار d_f داشته باشد، شاخه های دیگر مقدار ۱ میگیرند. یا اگر در یک شاخه مقدار ۰ بود، شاخه های دیگر و خطی که از آن مشتق شده نیز مقدار ۰ میگیرند.

تابع d_frontier شامل چک کردن شرط frontier بودن یک گیت است. میدانیم یک گیت در صورتی میتواند frontier تلقی شود که: ۱) نزدیک تر از سایر گیت ها به خروجی باشد و ۲) یک ورودی آن مقدار اشکال و دیگری نامشخص باشد.

در این تابع با یک حلقه و دو شرط این امکان بررسی میشود و نتیجه (D-frontier) برای تابع fault_prop بازگردانده میشود. همچنین، در طول اجرای انتشار خطا تابع d_frontier چندین بار فراخوانی میشود که در هر بار فراخوانی، گیت frontier بروز رسانی میگردد.

۲.۵. ارزیابی و درستی سنجی مقادیر اعمال شده (Consistency)

پس از آنکه مقدار اشکال به یکی از خروجی های مدار رسید، تابع consistency فراخوانی میشود تا مقادیر سایر نودها که هنوز مقداری نگرفته اند مشخص شده و بررسی گردد که آیا مقادیر اعمال شده به نودها در مرحله پیشین (prop_matrix) با مقادیری که اکنون بوسیله روش forward/backward implication بدست می آیند (cons_matrix) تداخلی دارند یا خیر.

در تابع consistency، ابتدا توسط لیست unvalued_data نودهایی که هنوز مقداری نگرفته اند مشخص میشود. این نود ها سپس بوسیله مقادیر بدست آمده در مرحله قبل، مقدار دهی میشوند. مقدار دهی در دو بخش ۱) مقدار دهی عقب رو (backward implication) و ۲) مقدار دهی جلو رو (forward implication) انجام میگیرد.

در بخش مقدار دهی عقب رو، توسط حلقه ای از نود های انتهای مدار به نود های ابتدا می آییم و بررسی میکنیم که آیا این نود در مرحله قبل مقدار گرفته است یا خیر. اگر مقدار آن در مرحله قبل ثبت شده و مقداری قطعی است (یعنی میتوان ورودی ها را بدون نیاز به دانستن سایر نود ها مقدار دهی کرد) ورودی های بدون مقدارش، مقدار دهی میشوند. مثلا اگر گیت AND باشد و خروجی آن ۱ باشد، قطعا ورودی های آن تماما باید مقدار ۱ داشته باشند. سپس نود های مقدار گرفته از لیست unvalued_data حذف میشوند.

در بخش مقدار دهی جلو رو، همانند فاز ۱ پروژه، خروجی گیت هایی که مقدار ورودی آنها مشخص شده را بدست می آوریم و آنها را از لیست unvalued_data حذف میکنیم.

پس از مشخص شدن مقادیر تمامی نودها، ورودی هایی که مقدار قطعی ندارند (یعنی ۱ یا ۰ بودن برایشان تفاوتی نمیکند) مقدار x میگیرند.

در نهایت از روی cons_matrix که شامل نود هایی هستند که در مرحله consistency بدست آمده اند، دوباره مقادیر بدست آمده از مرحله انتشار خطا را محاسبه کرده و اگر تداخلی میان مقادیر قبلی و مقادیر جدید دیده شد، نشان دهنده وجود inconsistency میباشد و باید وارد مرحله backtrack شویم.

۲.۶. عقب گرد (Backtrack)

همانطور که گفته شد، زمانی که مقادیر جدید برای نود ها بدست می آید و با مقادیر قبلی در تضاد است، باید مقدار آخرین ورودی را از پشته ورودی ها برداشته و آنرا تغییر دهیم. سپس دوباره به مرحله consistency بازگشته و مقادیر جدیدی را بدست می آوریم و با قبلی ها مقایسه میکنیم.

مرحله backtrack تا جایی که تداخلی میان مقادیر جدید و مقادیر قدیم وجود داشته باشد، اجرا میشود.

۲.۷. ذخیره بردارهای تست و اشکالات کشف شده در فایل

هنگامی که تمامی مقادیر نود ها، شامل بردارهای تست (مقادیر ورودی ها)، نودهای میانی و خروجی ها بدست آمد، به تابع get_fault_vec ارسال میشود. مهمترین متغیر این تابع، دیکشنری fault_vec است که کلید هر سطر آن یک بردار تست و مقدار آن، اشکال یا اشکالات کشف شده توسط آن بردار تست است.

این متغیر به همراه نام مدار به تابع fault_vec_done ارسال شده و در فایلی با فرمت fault_circuit_name.txt در پوشه fault ذخیره میگردد.

۳. فشرده سازی بردار های تست

در فایل اصلی اجرا (main.py)، آخرین تابعی که فراخوانی میشود، تابع compress_fault_vec میباشد که تنها یک پارامتر (نام مدار) را دریافت میکند.

در ادامه، ابتدا الگوریتم فشرده سازی بردارهای تست را بیان کرده و سپس وارد پیاده سازی آن میشویم.

۳.۱. الگوریتم فشرده سازی بردارهای تست

فشرده سازی بردارهای تست را میتوان به دو شکل ۱) درخت دودویی و ۲) درخت مورب پیاده سازی کرد. در روش اول که به الگوریتم تونومنت نیز معروف میباشد، دارای پیچیدگی زمانی $N \log N$ میباشد که در آن N، تعداد بردار های تست بدست آمده از الگوریتم D است. اما مشکلی که این الگوریتم دارد، عدم فشرده سازی تمام بردارهای تست میباشد. به عبارتی دیگر، این الگوریتم با توجه به ساختار درخت دودویی که دارد، نمیتواند تمامی بردارهای تست یکی پس از دیگری بررسی و مقایسه کرده و آنها را فشرده کند.

الگوریتمی که در این پیاده سازی از آن استفاده نموده ایم، همان روش دوم یا درخت مورب است. این الگوریتم پیچیدگی بالاتری از الگوریتم تونومنت دارد اما در عوض میتوان تمامی بردارهای تست را با هم مقایسه کرد و فشرده سازی را بطور کامل انجام داد.

پیچیدگی زمانی این الگوریتم از مرتبه $O(N^2)$ میباشد که در آن N، تعداد بردار های تست بدست آمده از الگوریتم D است. چگونگی بدست آوردن پیچیدگی این روش در قسمت بعدی، با توجه به کد توضیح داده خواهد شد.

۳.۲. پیاده سازی و کد فشرده سازی بردارهای تست

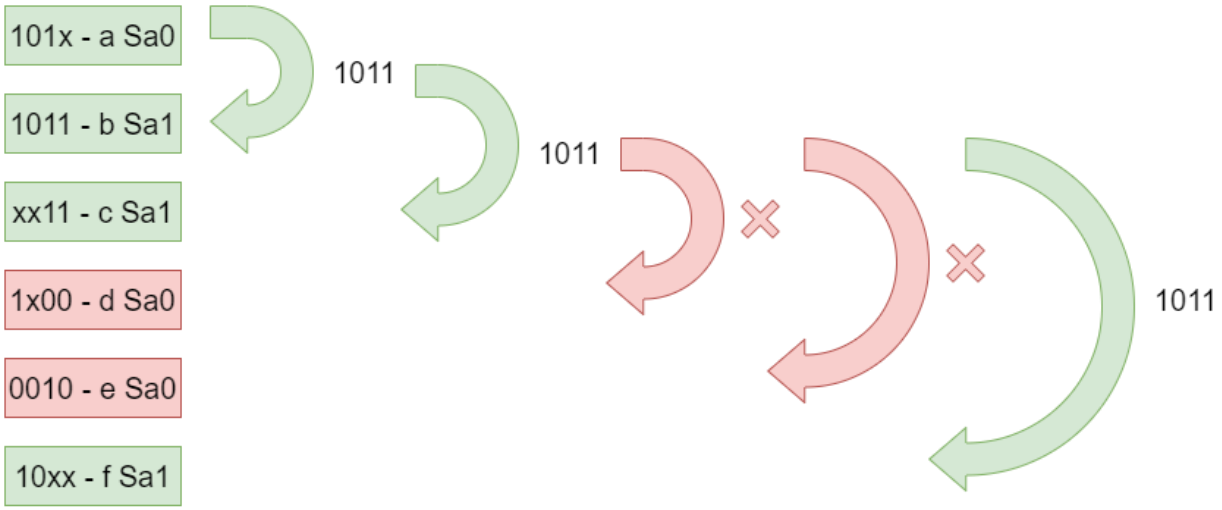
در تابع compress_fault_vec، ابتدا بردار های تست و اشکالاتی که هر کدام از آنها کشف میکنند را بوسیله خواندن فایل موجود در مسیر fault/fault_circuit_name.txt بدست آورده و درون دیکشنری test_vec قرار میدهیم (کلید ها برابر بردارهای تست و مقادیر متناظر با کلید همان اشکال کشف شده میباشد).

حال بوسیله دو حلقه در هم، به فشرده سازی بردارها میپردازیم. شمارنده (i) حلقه اول، از ابتدای لیست بردار های تست شروع شده و تا انتها میرود. شمارنده حلقه دوم (j) از یکی پس از شمارنده حلقه اول شروع شده و تا انتها میرود.

درون حلقه دوم، به ازای هر دو بردار تست، تابع compressor فراخوانی میشود. این تابع، کاراکتر به کاراکتر، تمامی ارقام موجود در دو بردار تست ورودی را با هم مقایسه کرده و به مقدار x حساس میباشد. اگر هر کدام از کاراکتر ها مقدار x داشته باشد، کاراکتر دیگر تعیین کننده مقدار نهایی است. همچنین اگر در هر کدام از کاراکتر ها، مقدار یک ورودی ۰ و دیگری ۱ باشد، این دو ناسازگار فرض شده و این دو بردار تست امکان فشرده شدن

ندارند و اجرا برای این شمارنده تمام میشود (return میشود). این کار برای تمامی شمارنده های حلقه اول اجرا میشود و در نتیجه تمامی بردار های تست با تمامی بردارهای دیگر مقایسه میگردند.

یک نمونه از این روش در شکل زیر آورده شده:



با توجه به وجود دو حلقه درون هم (nested loops)، در بدترین حالت تمام بردارهای تست با تمامی بردار های دیگر مقایسه شده که هزینه هر سری مقایسه N و در مجموع هزینه کل برابر $O(N^2)$ میباشد.

۴. مقایسه روشهای تولید بردار تست

جهت مقایسه زمان اجرای الگوریتم های D، تست جامع و فشرده سازی بردارهای تست، در فایل اصلی (main.py) از سه متغیر زمانی استفاده میکنیم:

۱) tota_test_time: زمان اجرای تست جامع
۲) d_algo_time: زمان اجرای الگوریتم D
۳) comp_time: زمان اجرای فشرده سازی بردارهای تست

این مقادیر از تفریق زمان پایان فرآیند مربوطه از شروع فرآیند بدست می آید. برای مثال، جهت بدست آوردن زمان الگوریتم D، از شروع تابع equivalence_faults_check تا پایان تابع fault_vec_done زمان گیری انجام میدهیم و آنرا در کنسول چاپ میکنیم.

نتایج در جدول ذیل آمده است:

*مدار کوچک: c17.isc

*مدار بزرگ: full_adder.isc

نام الگوریتم و مدار محک تست شده	زمان تولید بردارهای تست	زمان فشرده سازی بردارها	زمان اعمال بردارهای تست	زمان تست برای N مدار
D algo (c17)	0.01 sec	0.006 sec	0.19 sec	**0.016 + 0.19(N) sec**
D algo (full_adder)	0.1 sec	0.08 sec	0.52 sec	**0.18 + 0.52(N) sec**
Full test (c17)	***	***	0.33 sec	**0.33(N) sec**
Full test (full_adder)	***	***	0.53 sec	**0.53(N) sec**

۴.۱. نتایج مقایسه الگوریتم های تولید بردار تست

با توجه به نتایج بدست آمده از جدول فوق، نتایج زیر بدست می آیند:

۱) در مدارهای محک با سایز بزرگ (مثل full_adder)، تست جامع و الگوریتم تولید بردار تست D زمانی مشابه دارند. دلیل این امر، پیچیدگی الگوریتم D در قسمت های انتشار اشکال و باز تولید مقادیر نود ها در مرحله consistency است که زمان زیادی صرف میکند. اما در روش تست جامع، تمامی بردارها از ابتدا وجود دارند و تنها آنها را به مدار اعمال کرده و نتایج را در خروجی مشاهده میکنیم.

۲) در مدارهای کوچک (مثل c17)، الگوریتم D عملکرد بسیار بهتری نسبت به تست جامع دارد. زیرا اولا در الگوریتم D، ابتدا تعداد زیادی از اشکالات توسط هم ارزی اشکالات حذف شده و تعداد باقیمانده سریعا در خروجی شناسایی میشوند.

۳) زمان فشرده سازی بردارهای تست، طبیعتا با تعداد بردارهای تست ورودی رابطه مستقیم دارد. به عبارتی دیگر، با افزایش سایز و تعداد بردارهای تستی که فشرده سازی به عنوان ورودی دریافت میکند، زمان آن به شکل نمایی افزایش می یابد (با توجه به پیچیدگی $O(N^2)$)

ساختار پروژه:

فایل ارسالی یک فایل زیپ شامل گزارش فاز ۱، گزارش فاز ۲ (ورد و پی دی اف) و پوشه project_code میباشد.

**فایل های تست شده: ورودی آنها در پوشه input، خروجی آنها در پوشه output و فالت آنها در پوشه fault

ساختار پوشه project_code:

۱) پوشه fault: شامل نتایج بدست آمده از الگوریتم D (بردارهای تست تولید شده و اشکالی که هر بردار بدست می آورد) و نتایج بدست آمده از فشرده سازی بردارهای تست.
فرمت: comp_circuit_name.txt fault_circuit_name.txt

۲) پوشه input: فایل های ورودی باید در این پوشه قرار گیرند. این فایل شامل دو ستون input و value میباشد که ستون اول شماره ورودی مدار و ستون دوم مقدار آن ورودی است.
فرمت: input_circuit_name.txt

۳) پوشه output: فایل های تولید شده به عنوان خروجی مدار در این پوشه قرار میگیرند. این فایل شامل دو ستون node و value میباشد که ستون اول شماره نود در مدار و ستون دوم مقدار آن نود است.
فرمت: output_circuit_name.txt

۴) پوشه isc_sample_files: شامل فایل های مدارات محک میباشد.

۵) فایل main.py: ورودی کد و فایل اصلی قابل اجرا.

۶) فایل handler.py: توابع مهم و اساسی پیاده سازی شده.

۷) یک نمونه از چگونگی اجرا و کنسول در فایل guid.png آورده شده.